O'REILLY®

TURING

Hello, Startup

奔跑吧，程序员



从零开始打造产品、技术和团队

[美] 叶夫根尼·布里克曼 ○著　吴晓嘉○译

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

延展阅读

《进化：从孤胆极客到高效团队》

978-7-115-43418-0

Brian W. Fitzpatrick、Ben Collins-Sussman 著，金迎 译

- 技术人员版《人性的弱点》
- 提升职业生涯软技能
- 探讨领导力、合作、沟通、高效等团队成功关键因素

《用户思维+：好产品让用户为自己尖叫》

978-7-115-45742-4

Kathy Sierra 著，石航 译

- 颠覆以往所有产品设计观
- 好产品 = 让用户拥有成长型思维模式和持续学习能力
- 极客邦科技总裁池建强、公众号二爷鉴书出品人邱岳作序推荐，《结网》作者王坚、《谷歌和亚马逊如何做产品》译者刘亦舟、前端工程师梁杰、优设网主编程远联合推荐

《学习敏捷：构建高效团队》

978-7-115-44755-5

Andrew Stellman、Jennifer Greene 著，段志岩、郑思遥 译

- 精讲精益、Scrum、极限编程和看板方法
- 全面解读敏捷价值观及原则
- 提高团队战斗力

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

奔跑吧，程序员：从零开始打造产品、技术和团队

Hello, Startup

[美] 叶夫根尼·布里克曼 著
吴晓嘉 译

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（ＣＩＰ）数据

奔跑吧，程序员 ：从零开始打造产品、技术和团队 /
(美) 叶夫根尼·布里克曼 (Yevgeniy Brikman) 著 ; 吴
晓嘉译. -- 北京 : 人民邮电出版社, 2018.7
ISBN 978-7-115-48366-9

Ⅰ. ①奔… Ⅱ. ①叶… ②吴… Ⅲ. ①程序设计
Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2018)第087212号

内 容 提 要

本书以软件工程师出身的创业者角度，全面介绍了创业公司该如何打造产品、实现技术和建立团队，既是为创业者打造的一份实用入门指南，又适合所有程序员系统认识 IT 行业。书中内容分为三部分——技术、产品和团队，详细描绘创业的原始景象，具体内容包括 ：创业点子、产品设计、数据与营销、技术栈的选择、整洁的代码、软件交付、创业文化、招兵买马，等等。

本书适合所有程序员，尤其是准备创业的技术人员。

◆ 著　　[美] 叶夫根尼·布里克曼
译　　吴晓嘉
责任编辑　朱　巍
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：700×1000　1/16
印张：23.5
字数：549千字　　　　2018年 7 月第 1 版
印数：1 – 3 500册　　　　2018年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-0538号

定价：99.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

中文版推荐序一

这本书的文稿躺在我邮箱里好几天没有去看，主要原因是个人不太喜欢这个书名。在我的想象中，这又是一本攒出来再蹭蹭娱乐节目热点的平庸之作。但是当我想起来，真正打开书稿开始阅读的时候，我惊讶地发现跟我想象的完全不一样。

对，这就是想象世界和真实世界的差别。

程序员或者泛产品技术群体，是非常善于构建一个想象世界的，这也就是第一行代码为什么要写“hello, world”。但是当想象世界需要在真实世界落地的时候，就会产生很多的冲突和不知所措。

因为在写代码的时候，结果都是非常确定性的，编译成就是成，不成就是报警。而现实的创业生活中，有太多的灰色地带，你分不清楚对错，这时候就特别需要有人能告诉你，他当时遇到这样的问题，是怎么做的。

在和脱不花、罗胖一起创立罗辑思维和得到的四年时间里，我们已经记不清有多少次三个人面面相觑，慨叹“创业进入深水区，不会玩了”。然后下一次，我们会发现，上一次的深水区，压根儿才到脚脖子，这次才是真正淹到了脖子。

我们所能做的，就是遍访身边的创业者，问问他们遇到同样的问题怎么解决，过程苦不堪言。而当我看到这本书的时候，眼前不由得一亮，这其实就是把其他人走过的坑，变成你的活地图。如果这本书能早两年出来，就能让我避免走很多弯路。

在想象的世界里，因为你根本没有接触过这个世界的其他方面，很多技术型创业者连沙盘推演都不知道到底有多少“蓝军”。我看到很多优秀的技术人员，在创业中，没有在技术上失败，却败在了产品上、市场上、财务上、税务上、团队管理上……简直就是九死一生。

这本书肯定无法解决很多创业中的实际问题，但是它提供的很多方法论，是可以帮助年轻的技术创业者提前预见可能会遇到的问题的，要么跳过坑，要么换条路。

创业是件很痛快也很痛的事情，上路不易，带张地图吧。

——快刀青衣，罗辑思维 & 得到联合创始人

中文版推荐序二

奔跑的命运

《奔跑吧，程序员》这个名字，让我想到一部电影，《罗拉快跑》。

在《罗拉快跑》里，罗拉需要在 20 分钟内弄到 10 万马克，否则男友就会死于非命。电影用平行结构，展示了三个不同的结局，让观众看到罗拉和男友三种可能的命运。

身为观众，我们当然可以看到三种结局，但如果我们就是罗拉，我们只能有一种宿命。

创业也是这样。创业公司成百上千，命运也成百上千。我们当然可以艳羡那些成功创业者，但如果自己创业，如何才能把握自己的命运？

《奔跑吧，程序员》虽然不能保证皆大欢喜的创业结局，起码可以避免出现悲剧。全书主要从三个方面展开。第一，产品。怎样选择合适的点子，怎样把它打造成产品，怎样验证产品的有效性和价值。第二，技术。怎样选择技术栈，怎样设计架构，怎样交付。第三，团队。怎样组建和激励团队，怎样找到靠谱的创业团队，怎样持续学习。

与通常泛泛而谈的“技术创业”书不同，《奔跑吧，程序员》对每个方面都给出了足够接地气的建议。举个简单的例子，“远程办公”的利弊已经有无数讨论了，但大多数讨论都着眼于形式。《奔跑吧，程序员》直截了当地指出：真正的问题不在于“远程办公”。即便是集中办公，两三个人在咖啡机旁讨论完，并不把过程和结果同步给需要知道的所有人，这种配合方式还不如“远程办公”。

然后作者进一步指出，无论“远程”还是“集中”，要保证效率和健康，团队都应当以“分布式”的方式协作，电子化、公开化、不加锁，同时要确保大家交流不能“就事论事”，一定要有“聊出某些新鲜点子”的机会。这道理看似简单，但仍然有许多创业团队不明白，在这类问题上犯了想当然的错误，结果创业失败，甚为可惜。

《罗拉快跑》的一个结局是，罗拉在赌场赢了钱，男友曼尼找回了钱袋，这个结局堪称“皆大欢喜”。我衷心希望，《奔跑吧，程序员》的读者在创业路上能找到皆大欢喜的结局。

——余晟，“余晟以为”微信公众号主人

中文版推荐序三

创业是人生最好的修炼

近期发生的中美贸易战，暴露了中国企业在基础研发和创新方面的巨大短板。中国人与美国人相比，最欠缺的就是创造力。我说的是那种能够落地的创造力，而不是天马行空纯空想的创造力。

对于中国最有创造力的群体之一 ——程序员来说，创业是激发自己创造力的最佳途径。大数据、物联网、人工智能、区块链等技术的发展和普及，让现在成为程序员创业的黄金时代。而且对于中国程序员来说，现在是难得可以与美国程序员同步竞争的黄金时代。中国企业甚至有可能在上述这些领域后来居上。

《奔跑吧，程序员》这本书恰逢其时，正是一本专门面向程序员的创业指导书。虽然曾经创业过多次，这本书仍然带给我相见恨晚的感觉，读后受益匪浅。如果你有志于走上创业道路，一定要提前读一下这本书。创业是一件非常艰难的事情，需要具备很多方面的能力。如果你没有做好准备就贸然上路，必然会劳而无功，白白浪费大量时间和金钱。

歌曲《在人间》中有这样一句歌词：“在人间，有谁活着不像是一场炼狱。”确实，人生就是一场巨大的磨难。非常幸运的是，我们是程序员，我们其实有很多创业的机会。人生在世，至少要有一次创业的经验，才可以说不枉此生了。我们可以把创业作为人生最好的修炼，创业的经验可以让我们变得真正强大起来。

——李锟，上海霓风网络科技有限公司 CEO

中文版推荐语

这本书是程序员技能扩展的好书，在代码上有所建树的程序员很多会自然而然追求更高的挑战：如何带领一支队伍打造一款不错的产品。这本书对想独立创业的程序员来说更具有参考意义。当然我是建议程序员们都读读这本书，至少能知道代码之外原来还有这么多可以去思考琢磨的东西。

——余弦，慢雾科技联合创始人

技术型创业不仅仅需要勇气，还需要有正确的方法。本书通过产品、技术和团队三个方面的经验介绍，让尝试创业的程序员可以少走一些弯路。作者提出的“伟大的公司是进化而来的”观点，值得我们这些程序员创业者去深刻体会和理解，能够帮助我们穿透迷雾找到正确的企业发展方向。

——李启雷，趣链科技联合创始人兼首席技术官

如果你正在寻找一本科技创业权威指南或想深入理解科技创业，那么本书无疑是你最佳的选择之一。作者通过产品、技术和团队这三个维度，全方位系统性地阐述了科技创业方法论。无论你是投资人、创始人、高管，抑或是项目经理、开发者和大学生，这本书都将对你的职业生涯有所裨益。

——彭靖田，谷歌机器学习开发专家，《深入理解 TensorFlow》作者

结合自己的两次创业经历，我深知创业就是一个不断踩坑和挖坑的过程。虽然始终免不了要犯错，但哪怕少一个错误，对基础薄弱的创业公司的存活都是至关重要的。这本书提到了很多避免犯错的技巧，值得每一个创业者学习。

——徐谦，贝米钱包 CTO

创业圈流行一句玩笑：就缺一个程序员了。那我们程序员创业缺什么呢？从构建团队到产品运营，这些技术之外的能力都需要我们一点点补全。而这本书就像一位创业教练，会手把手教你实战，从而极大提高你成功的概率。电子世界的魔法师们，共产主义要靠你我的奋斗，去创造属于自己的时代传奇吧！

——袁行远，彩云科技 CEO，彩云天气与彩云小译程序员

对本书的赞誉

对创业技术团队而言，良好的计算机科学教育与所谓的“常识”之间存在着巨大的鸿沟。大多数人都必须通过博客、工作，当然还包括学校的磨练才能学到这些知识，本书涵盖了许许多多这样的“至理名言”。我真希望自己在入行的时候就能遇到这样一本书。

——Jay Kreps，Confluent 公司 CEO

作为创业者，你肯定希望只花一点时间，就能学到纷繁复杂的多个学科的大量知识。虽然我很珍惜创业时不断碰壁又突出重围的经历，但还是希望自己开始时就能有这样一本指南。

——Bowei Gai，CardMunch 创始人、CEO

Jim 以广阔的视角带你了解软件创业需要掌握的方方面面。本书不是由术语堆砌而成的，也不是空洞之谈，只是实实在在、简单且被证明可行的建议——这就是我读本书初稿时的直观印象。如果你曾有过“我该如何想出创业点子”“我应该在这个项目中用什么技术”或者“我怎么被那些了不起的创业公司录用”这样的疑问，本书无疑就是为你准备的。

——Eugene Mirkin，Array 风投公司企业家

和 Jim 一样，我的职业生涯也是从大公司开始的。现在我是自己的公司 prismic.io 的联合创始人，我每天都能从这段经历中学到很多。本书正是把许许多多这样的知识呈现给大家。它不但阐明了如何通过创业去释放自己的真正潜力，更告诉你这样做的原因所在。

——Sadek Drobi，prismic.io 联合创始人

如果所有的计算机系都把本书作为毕业礼物送给学生，科技行业将会发生两件好事：最糟糕的科技公司将关门歇业，出色的那些则会明显变得更好。

——Brent Vince，Adacio 公司创始人

本书可谓是独一无二的一本介绍如何创业的实践之书，阅读体验也非常好。我真希望自己当初踏上创业之路时也能有这样一本宝典。

——Sean Ammirati，Birchmere 风投公司合伙人

献给妈妈、爸爸、Lyalya 和 Molly。

目录

第二部分 技术

第三部分　团队

前言

```
main( ) {
  printf("Hello, World");
}
```

我们在学习一门全新的编程语言时，最先看到的往往是一份“Hello, World”教程，教你如何把文本“Hello, World”显示到屏幕上，由此了解怎样让一个基本程序运转起来。而本书就像一本给创业者的“Hello, World”教程，将告诉你如何打造产品、开发技术和组建团队的。

我真希望自己在上大学时就能读到这样一本书，因为我虽然获得了学士和硕士学位，也有过不少实习机会，但对自己所做的事还完全没有概念。

我自己早期做过一些大项目，比如在汤姆森金融公司（Thomson Financial）开发过一个用于性能测试的桌面应用程序。那时我根本不知道怎么做用户界面，所以就随便把一些文本框、菜单和按钮放到界面上；对如何处理性能问题更是一头雾水，也就是在代码中随意加上一些缓存和线程池；同样，我也不懂得要考虑代码的可维护性，根本就没去操心测试和文档的事情，反倒是把几千行代码都塞入一个巨大的文件中。

我在 TripAdvisor[1] 的第一个项目则是为一个网页添加一些新的选项，该网页可以列出一个城市的所有酒店。这只是一个简单的任务，就是公司为了让我熟悉一下代码库而已。我在第一个星期就完成了任务，把网页推送到生产环境中。过了没多久，我就被叫到经理办公室，和经理来了个一对一的面谈。我看着他在网页上点开巴黎的酒店列表，选中我加进去的那个新选项，然后就开始等啊等啊，那个页面差不多花了“两个小时”才加载完。好吧，实际可能就是两分钟不到，但我真真切切地感受到，在狭义相对论的作用下，当一个人大汗淋漓恨不得找个地洞钻下去时，时间肯定发生了膨胀。那天晚上，我一直到深夜才发现那段花哨的新代码在进行排序时，每次比对酒店都要调用两次数据库，所以如果要对 n 项进行排序，大概需要进行 $O(n \log n)$ 次比较。巴黎有将近 2000 家酒店，页面加载一次可能会引发将近 40 000 次数据库调用。那天我虽然没有钻到地底下，但我们的数据库服务器可能差不多要累趴下了。

注 1：TripAdvisor 是一家国际性旅游评论网站，提供饭店、景点、餐厅等世界各地相关旅游资讯，官方中文名为“猫途鹰”。——译者注

我不会忘记那段时间出现了多少烦人的问题、丑陋的代码和难看的用户界面，也不会忘记，我经历了多少次网站宕机和多少个不眠的夜晚。但是，最让我耿耿于怀的却是问题这样多，我还找不到答案。比如，我该学习、使用什么技术？为什么我还要费心去考虑自动化测试？怎样才能做出一个不惹人厌的产品？怎样才能让别人使用我的产品？面对工作机会时，我该怎么进行谈判？是要争取更多的薪水还是更多的股权？股权究竟又是什么？我是应该在大公司工作，还是该加入创业公司？

对于上述问题以及其他种种问题，我费了很大的力气才找到了答案。所以，我也尝试着把自己学习到的东西（大部分都是经历了痛苦的磨难和犯了错误之后才得到的）用博客记录下来，或者通过演讲和他人分享。但在意识到有成千上万的开发者也会有同样反复试错的经历之后，我觉得是时候做一些更实质性的事情了。于是，也就有了这本书。当然，有些事情是自己犯过错才能学到的，但除此以外，我希望本书可以让读者从他人的错误中吸取经验，避免重蹈覆辙。

我觉得自己犯过最大的错，就是在职业生涯的早期对创业缺少关注。我的前几份工作都是在知名的大机构（思科、汤姆森金融、康奈尔大学），后来才偶然跳到创业公司（LinkedIn、TripAdvisor）。结果，在这些公司的所见所闻让我惊讶不已。我在创业公司前几个月学到的东西，比之前工作、实习和在学校的那些年加起来还要多。

创业公司并不仅仅是大公司的简化版，就像量子力学并不只是经典力学的简化版那样简单。经典力学描述了宏观物体（比如棒球或星球）以可预见和确定的规则在相对低速的运动中表现出的行为。与之类似，大公司通常都生存在具有确定规则的环境中，它们行动缓慢，因为它们面对的客户和产品都是可知的。量子力学则描述了微观粒子（比如光子和电子）基于某些不可预知、非确定的规则在极高速的状态下运动时所表现出来的行为。同样，创业公司经常在一无所知而又变化莫测的环境中高速运转。虽然很多人都了解经典力学和大公司，但只有对量子力学和创业公司也同样了解，才能看清世界的全貌。由此可见，我们必须从全新的角度去考虑问题，必须用全新的方法去工作。

我们工作的方式很大程度上其实也就是生活的方式，因为我们有一半的清醒时间是用来工作的。难道你不希望用这些时间去做一些可以让自己快乐的事情吗？我曾经认为，从事软件相关工作的人们面对的都是一望无际的小隔间、无能的老板、一份份 TPS（测试过程说明书）报告和各种企业代码。幸好，世界还有另外一面，这就是本书要向你展示的。我会介绍当今世上最出色的创业公司是如何工作的，让你了解这另外的一面。我相信，即便你从来没想过要加入创业公司，这些公司的理念对你也是有所裨益的。随着创业越来越普遍[2]，这些理念也同样会越来越有用。

本书源于我自身的经历和大量的研究，其中就包括对一些程序员的访谈，他们均来自过去十年最成功的创业公司，比如 Google、Facebook、Twitter、GitHub、Stripe、Instagram、Coursera、Foursquare、Pinterest 和 Typesafe（受访者的完整名单见“访谈”）。纵览本书，到处都闪烁着这些人的故事和思想。他们向我们描绘了创业生活的原始景象，不掺杂任何自我营销和公关——单纯就是程序员们在分享自己获得的成功和犯下的错误，并给出他们的建议。

注 2：仅仅美国一地，每个月就诞生将近 50 万家小型企业，它们提供了超过 66% 的新就业岗位。

本书的内容

我为这本书定下的目标是为创业公司打造一份既实用又具有可操作性的入门指南。这本书包含三部分内容：产品、技术和团队。接下来，我会列出每一部分都包含哪些章，并概括介绍每章具体讲解的技术、工具和技巧。

第一部分：产品

第 1 章　为何创业

为何打造面向大众的产品，如今（超越了历史上的任何时候）唯有创业才能为我们创造最佳的机会；什么是创业公司；是什么原因让你选择在创业公司工作，又是什么原因使你放弃在创业公司工作。

第 2 章　创业点子

如何想出创业点子；介绍点子日记、约束条件和痛点的概念；想法和执行的对比；博伊德迭代法则；如何利用客户开发过程快速、低成本地验证自己的想法。

第 3 章　产品设计

介绍大家都应当掌握的设计技能；用户界面应当如何设计才不会让用户觉得自己愚蠢；以用户为中心的设计原则（人物角色、情感设计、简单、可用性测试）；视觉设计的原则（文案、设计重用、布局、排版、对比与重复、颜色）；介绍如何设计最简可行产品（MVP）。

第 4 章　数据与营销

介绍每个创业公司都应当采用的度量指标；数据驱动的产品开发方法；A/B 测试；为什么最出色的产品未必能够胜出；推广、病毒式增长以及创业公司的销售策略。

第二部分：技术

第 5 章　技术栈的选择

应该利用内部资源开发软件，还是购买商业产品或使用开源产品；最初的技术栈应该如何选择；如何进化技术栈和重写代码；如何评估编程语言、框架和数据库。

第 6 章　整洁的代码

为什么说程序员的工作并不是写代码，而是理解代码；为什么代码的编排布局、命名、错误处理、不写重复代码原则（DRY）、单一职责原则（SRP）、松耦合、高内聚等可以让代码更容易理解；函数式编程为什么可以让代码易于重用；为什么重构对于编写良好的代码必不可少。

第 7 章　可扩展性

创业公司应当如何调整，才能适应更多的用户和开发人员；怎样修改代码才无须担惊受怕；如何应用测试驱动开发（TDD）获得更好的代码；如何在创业公司中引入设计评审、结对编程和代码评审；为什么说明文档是代码库中最重要的文档；如果你无法测量，就无法解决；如何利用估算推断性能状况。

第 8 章　软件交付

编写完代码之后要做的事情；为什么要使用源代码控制、开源构建系统和持续集成；如何进行配置管理、自动化部署和持续交付；如何为代码增加日志、监控和警告功能。

第三部分：团队

第 9 章　创业文化

为何要明确公司的使命和价值；管理层级架构和组织扁平化之间的衡量取舍；公司文化在人员招聘、晋升和激励中的作用；如何为程序员设计理想的办公室；远程办公的衡量取舍；创业公司的沟通策略和方法。

第 10 章　求职之路

如何利用人脉找到在创业公司工作的机会；如何让简历受到关注；如何才能在面试中有出色的表现；如何才能做好白板编程；如何才能提出好问题；在薪水和股权的问题上应该如何考虑；面对工作机会，应该如何谈判。

第 11 章　招兵买马

为什么说人才是创业公司最为重要的因素；创业公司要招什么样的人（合伙人、早期员工、通才和专才）；如何找到出色的候选人（以及如何打造公司品牌去吸引人才）；白板编程为什么是一种糟糕的面试方法（应当采取什么替代方法）；如何给出让人无法拒绝的录用条件。

第 12 章　学习

世界上最引人关注的软件开发者；为什么要撰写博客、文章、论文和图书；为什么要在小组会议、技术演讲和学术会议上发言；为什么几乎所有的代码都应该开源；为什么应该分享自己所知的几乎所有东西。

重要观点

除了上面所说的技术、工具和技巧之外，有三个重要的观点会在整本书中不断地出现，它们都是成功创业所不可或缺的，这三个观点分别是：创业与人密不可分、伟大的公司是进化而来的、速度制胜。

创业与人密不可分

> 工作面临的主要问题与其说与技术有关，不如说本质上属于社会学的范畴。
>
> ——Tom Demarco、Timothy Lister，《人件》

我们在课堂和书本上学到的有关创业的大部分内容，比如营销计划、产品设计、系统设计、测试策略、招聘计划和组织结构等，其实只是创业过程中的必然产物。仅仅研究这些产物，无法对创业有充分的理解。这就好比被铁链锁在柏拉图洞穴中的囚犯[3]，仅仅研究

注 3：这是柏拉图在《理想国》中设计的一个洞穴寓言，暗喻未受过真正教育、不愿面对真实世界的人。——译者注

面前墙上的影子，根本没有办法完全理解外面的世界是何种景象。

希望本书能让你走出洞穴，不仅了解创业的产物，也能了解创造这些产物的人；不仅学到如何设计出伟大的产品，也学到如何设计出以人为本的产品；不仅学到如何编写有效的自动化测试，也知道为什么有了自动化测试，就可以无须在修改代码时担惊受怕；不仅可以学到伟大的公司是如何组织的，更能学到为什么打造一家伟大公司最重要的是要懂得如何去发现和激励正确的人。

伟大的公司是进化而来的

> 有效的复杂系统一定是从有效的简单系统进化而来的。
>
> —— John Gall

看到长颈鹿的脖子时，你要知道这么长的脖子并不是老天爷一开始就故意设计的。随机突变导致一些长颈鹿的脖子变长，这又恰好提高了它们在某种特定环境下的生存概率，所以成千上万代之后，长颈鹿的脖子就变得越来越长。同样的道理，当我们见到一家成功的公司，必须认识到它的成功并不是创始人在建立公司的时候就计划好的，大多数创业公司都要历经数千次尝试才能有所改变并成长起来，归根结底不过是其中的一些尝试恰好提高了公司在特定市场中的生存概率，只是结果让人感觉是创始人在最早的时候就有所设计一般。

本书将会关注如何以一种增量、迭代式的发展方式去打造一家创业公司（好比敏捷和精益开发），而不是费心去找出完美的计划（好比“瀑布式开发”）。不管你是在打造产品、开发技术抑或建立团队，都会发现最好的起步方式其实就是先做出一个大概可以工作的最小的东西（最简可行产品，简称 MVP），然后再根据客户的反馈（对产品而言）、代码的评审和测试情况（对技术而言）或员工情况（对团队而言）逐渐进化，扩大规模。

速度制胜

> 世界正在快速变化，不再是以大胜小，而是以快胜慢。
>
> ——默多克

如果伟大的公司是不断进化且迭代发展的结果，那么迭代得最快的公司终将胜出。所以，本书的很多观点都和如何实现更快的迭代密不可分，也就是如何缩短反馈回路，加快学习步伐，进而提高进化的速度。客户开发有助于更快地发现合适的产品或市场；整洁的代码和自动化测试有助于更快地实现技术；强有力的文化则有助于更快地建立团队。另外，还有一个观点稍稍有悖于直觉，在后面你会了解到，做得更快，完成的质量会更好。所以说，速度致胜。

这是一本涉及广泛的书

本书所涉及的每个主题都已经有人写过书，甚至有多本书，所以我们只会关注最基本的概念，让你开启一段“Hello, World”式的学习体验。我们会推荐一些参考资料供你进一步学习。需要说明的是，本书并未涉及创业公司的法律和财务方面的内容。如果你对编

写商业计划、融资和上市等细节感兴趣，可以上网查阅相关书单。

毋庸置疑的是，仅靠阅读一本创业类图书是无法让你成为一名出色的开发者或公司创始人的，就好比阅读健身的书并不能让你成为出色的举重运动员。健身的书可以教会你一些特定的动作和练习，但在你第一次走到杠铃面前时，你仍然只是一只软脚蟹。只有花上无数的时间到健身房里锻炼、出汗，再应用从书中学到的知识，你才有可能举得起沉重的杠铃。同样的道理，本书的目的就是要教给你一些在创业中真正用得上的工具和技术，但也只有投入大量的时间去实践这些技术，你才有可能做到得心应手。

即便如此，书中所讲的东西也是没有对错之分的。有人说过，所有的模型都是错误的，但其中有一部分是有用的。书中所介绍的工具和技术在过去已经被证明对创业是行之有效的，我也希望它们以后对你也同样有用。但我们不要把这些东西当作现成的解决方案，而是要在自己的脑海中形成一系列的知识点，在思考问题的时候可以借此得到自己的解决方案。

本书面向的读者

如果你在创业公司中工作或者正打算投入创业大潮，抑或就职于大公司，却希望像创业公司一样去运作它，你应该读读这本书。本书将向你介绍如何在瞬息万变、科技创投变化无常的环境下，打造一家成功的公司（或一份成功的事业）所要掌握的所有基本概念。尽管这追根溯底是一本程序员写给程序员的书，但其实只有第二部分“技术”是明显偏技术的，第一部分“产品”和第三部分“团队”能被所有受众很好地理解。

如果你是一名刚入行的程序员，这就是一本适合你的书。本书可以说汇集了我在读大学时所有想知道的东西，也是我在刚开始工作时希望有人告诉我的建议、提示或诀窍。作为一名年轻的程序员，你应该已经掌握了两三门编程语言，也可能精通了几种库和框架，或者在学校里已经做过几个小应用。虽然你还没有做好什么准备，就已经有人会为你的这些技能买单，但你也很快就会知道自己在学校里学到的东西到底能不能很好地用在现实世界中。还是让我先告诉你答案吧：尚不够！那是一条艰难的道路，你也会重复之前的程序员所犯下的无数错误。当然，你也可以好好读读这本书，从第一天起就让自己的职业朝着正确的方向发展。

如果你是一名经验丰富的开发者，本书会让你对每天都在从事的工作有系统性的认识。你还在要求面试者在白板上遍历二叉树吗？在为自己最近的项目挑选技术时，依靠的还是自己的直觉或网上的最新热门趋势？你的待办清单上是不是有一项是“编写文档”？你是不是觉得自己的公司已经变得过于臃肿、行动太慢、缺乏创新？我相信书中的故事可以让你频频点头微笑，你也能够把部分建议应用到现有的工作中，也可能让你决定是时候做些改变了。

如果你是科技公司的经理、执行官或投资人，这本书将帮助你弄明白为什么有时候对时间的估算会和实际情况相差一个数量级，为什么手底下最好的开发者非要跳槽到另一家公司，又为什么最新的“敏捷极限结对 XXX”方法并没有让你的团队变得更加高效。你的成功在很大程度上取决于能否理解程序员的思维方式、弄清楚他们一天到晚到底在做什么，以及如何去激励他们。书中的故事可以说都是大白话，哪怕你和下属进行一对一的交谈，也是听不到这些话的。

如果你尚未投身创业，但已经开始对创业感兴趣，这本书其实就是一位知情者在告诉你事实的真相。如果只通过研究最终的产品（例如网站、移动应用、炫酷的小玩意儿）就想理解一家成功的创业公司，无异于只看一下某个人的毕业证书就想弄清楚他的大学经历——虽然那张纸的确令人印象深刻，但却无法透过它看到一个人为了得到它而去上课、参加研究会议、考试、做作业、经历成功与失败的那些岁月——而这一切都是必不可少的。LinkedIn 和 Facebook 这样的公司看似简单，但实际并非如此。本书会向你揭示，究竟这些公司内部有什么样的创新；它们如何去解决问题；又经历了多少不眠的夜晚，才使得这一切成为了现实。简而言之，只要你对创业感兴趣，这本书就适合你。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语或重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://bit.ly/Hello_Startup。

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：
http://facebook.com/oreilly

请关注我们的 Twitter 动态：
http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：
http://www.youtube.com/oreillymedia

致谢

本书的诞生得益于许多人的帮助。我最初之所以起了写书的念头，要归功于以下诸位的建议和帮助，他们是 Joe Adler、Adam Trachtenberg、Joshua Suereth 和 Nilanjan Raychaudhuri。感谢我的黑客朋友 Florina Xhabija Grosskurth、Matthew Shoup、Prachi Gupta 和 Bowei Gai，他们一路以来给我提供了很多想法和反馈。

O'Reilly（以及 O'Reilly 之外）的朋友们，特别是 Angela Rufino、Mary Treseler、Nicole Shelby、Gillian McGarvey 和 Mike Loukides，正是有了他们的帮助，我这样一个写作新手才能创作出值得出版的作品。另外，我还要感谢 Peter Skomoroch、Sid Viswanathan 和 Jiong Wang 为我引荐以及 James Yeagle 在法律问题上给予我的帮助。

有不少勇敢的志愿者看过此书的提前发布版，耐着性子去阅读未经雕琢、尚未准备就绪的内容，他们是：Alistair Sloley、Joseph Born、Clarke Ching、Jay Kreps、Ara Matevossian、Prachi Gupta、Matthew Shoup、Martin Kleppmann、Dmitriy Yefremov、David J. Groom、Molly Pucci、Steve Pucci、Alla Brikman 和 Mikhail Brikman。非常感谢你们能花时间来审阅我的作品，谢谢你们给予的反馈和帮助。

对于接受我访谈的了不起的程序员们，我更是要致以万分的谢意，他们是 Brian、Daniel、Dean、Flo、Gayle、Jonas、Jorge、Julia、Kevin、Martin、Mat、Matthew、Nick、Philip、

Steve、Tracy、Vikram 和 Zach（参见“访谈”）。没有你们的故事、建议和反馈，本书将处处碰壁，变得无趣而残缺不全。

最后我想说，我生命中的多数美好都是家人给予的。妈妈、爸爸、Lyalya 和 Molly，这本书是献给你们的。

访谈

作为本书研究工作的一项内容，我对过去十年一些最成功的创业公司的程序员进行了访谈。在和他们的讨论中，我了解到什么样的问题会在几乎所有的创业公司身上一次又一次地出现。这些谈话也让我有了一个更广阔的视角，得以了解不同的公司是如何考虑这些问题的，它们在解决此类问题时最常见的模式和实践方法是什么。由此我也深受启发，知道成为一名伟大的开发者需要具备什么样的条件。我把这些想法全部融于书中，并在书中的各个部分直接引用了以下访谈者的采访内容。

Brian Larson
: Google 资深软件工程师，Twitter 主任软件工程师。

Daniel Kim
: Facebook 软件工程师，Instagram 技术主管。

Dean Thompson
: Transarc 公司联合创始人，Premier Health Exchange 联合创始人、CTO，Peak Strategy 联合创始人、CTO，mSpoke 联合创始人、CTO，LinkedIn 技术总监，NoWait 公司 CTO。

Florina Xhabija Grosskurth
: LinkedIn 公司 Web 开发者、产品专家、经理，Wealthfront 人力运营主管。

Gayle Laakmann McDowell
: GareerCup 创始人、CEO，Seattle AntiFreeze 创始人、联席总裁，KeenScreen 有限公司技术副总裁，Google 软件工程师，《程序员面试金典》作者。

Jonas Bonér
: Triental AB 联合创始人、CTO，Scalable Solutions AB 创始人、CEO，Typesafe 联合创始人、CTO。

Jorge Ortiz
: Joberator 创始人，LinkedIn 软件工程师，Foursquare 服务器端工程师，Stripe 计算机专家。

Julia Grace
: WeddingLovely 联合创始人、CTO，Tindie 公司 CTO。

Kevin Scott
: LinkedIn 技术与运营高级副总裁，AdMob 技术、运营副总裁，Google 高级技术总监。

Martin Kleppmann
: Go Test It 和 Rapportive 联合创始人，LinkedIn 高级软件工程师。

Mat Clayton
　Mixcloud 联合创始人、CTO。

Matthew Shoup
　Indiaplaza.com Web 开发者，VNUS 医疗技术公司电商主管，LinkedIn 资深常驻计算机专家，NerdWallet 高级技术官（Principle Nerd）。

Nick Dellamaggiore
　LinkedIn 首席高级工程师，Coursera 基础架构设计领头人。

Philip Jacob
　StyleFeeder 创始人、CTO，Stackdriver 工程师，Google 资深软件工程师、技术经理。

Steven Conine
　Spinners 联合创始人、CTO，Wayfair 创始人。

Tracy Chou
　Quora 软件工程师，Pinterest 软件工程师。

Vikram Rangnekar
　Voiceroute 联合创始人，Socialwok 联合创始人，LinkedIn 资深软件工程师。

Zach Holman
　GitHub 早期招聘的一位工程师。

电子书

如需购买本书电子版，请扫描以下二维码。

第一部分

产品

第 1 章

为何创业

1.1 科技创业的时代

> 大约五亿四千万年前，地球上发生了奇妙的事情：生物的形式开始出现多样化，导致了所谓的“寒武纪生命大爆发”。在此之前，海绵动物和其他简单生物主宰着地球，但就在几百万年内，动物王国变得极为丰富多彩……类似的情况正在虚拟领域上演，我们正在经历“创业者大爆发”。数字式创业蓬勃发展，提供了超乎人们想象的服务和产品，渗透进了经济生活的方方面面。它们正在重塑各个产业，甚至改变公司这一概念。
>
> ——《经济学人》

此时此刻，在世界的某个角落，两个程序员正坐在车库里敲着一行行代码，创造着我们的未来。我们正处在高科技创业的时代，硅谷是引领者，但每一个主要城市，从博尔德到伦敦，从特拉维夫到新加坡，都在试图打造自己的创业中心。仅仅在美国，就有 1000 多家风险投资公司和 20 万名天使投资者，每年对初创企业的投资大概在 500 亿美元左右。2010 年，美国的创业者成立了 3 万多家高科技和通信技术公司，几乎每小时就会诞生差不多 4 家科技创业公司。

创业革命就在眼前。本章将解释为什么创业是我们必须关注的事情。（阅读本书就是一个好的开端！）我将探讨创业之所以伟大的一些原因，以及为什么我们应该考虑加入创业公司，甚至自己亲自创业。实话实说，在讨论中我也会承认，创业确实有它令人生厌之处，还会探讨为什么不是每个人都适合创业。但首先，我要对书中所说的“科技创业公司”下个定义，因为这个词对不同的人有不同的含义。

1.2 什么是科技创业公司

本书主要关注的是**科技创业公司**。“科技”很好理解，如果公司业务发展主要依赖于技术

的**研发**——这里的技术不管是实际销售的产品，还是用来销售其他产品的技术——那么它们就是科技公司。如果公司主要**使用**的是现有的技术，那么就算不上是科技公司。例如，GitHub 之所以是一家科技公司，是因为它在研发和销售一些可以让程序员更容易相互协作的技术。同样，TripAdvisor 也是一家科技公司——虽然销售的是旅游产品（例如酒店房间、度假套餐、机票），但为了达到这一目的，员工的大部分工作是去技术研发，比如酒店网页、用户账号、评论存储、照片存储和搜索功能。而一家本地的餐厅就谈不上是科技公司了，哪怕这家餐厅有一个精美的网站，哪怕该网站是用 Flash 写的，还可以自动播放音乐。这是因为该餐厅的主要经营活动是为用餐者提供美食和良好的氛围，而不是提供技术。

说清楚了“科技”这个词，我们再来看看“创业公司”这个词。典型的创业公司就是两个开发者上个星期刚在车库中建立的公司，但是“创业公司”这个词有时也用来描述一些更大、成立时间更长的公司。例如，华尔街日报把“创业公司”这个词用在下面这些公司身上。

- SnapChat：估值 100 亿美元，成立 2 年，雇员 20 多人。
- Uber：估值 420 亿美元，成立 5 年，雇员 550 多人。
- SpaceX：估值 48 亿美元，成立 12 年，雇员 3000 多人。

可以看出，创业公司的定义并不是看它值多少钱（从 0 到 420 亿美元），也不是看它建立了多长时间（从 1 周到 12 年），或者看它拥有多少员工（从 3 到 3000 名）。那么，什么是创业公司呢？要回答这个问题，我们还是先看看一些知名创业者给出的定义，先从 Eric Ries 开始。

> 创业公司就是在极度不确定的条件下创造新产品或服务的人类组织。
>
> ——Eric Ries，《精益创业》

创造产品和服务这点很好理解，创业确实也要面对大量的不确定因素，但大多数的本地餐厅也面临着同样的情况，它们失败的概率和绝大多数的创业公司类似。不过，我们通常都不会把本地的比萨店称为创业公司，所以还需要更进一步的定义。来看看 Paul Graham 是怎么说的。

> 创业公司的目标在于快速增长。一家公司成立的时间短并不能让其本身成为创业公司，创业公司也未必要从事科技领域的工作，未必要接受风险投资基金或有某种“退出”的机制。创业公司唯一必不可少的东西就是增长，其他和创业相关的所有东西都是伴随着增长而来的。
>
> —— Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

除了不确定性之外，我们现在知道创业公司又多了另一项必备的要素：大幅增长。本地比萨店的目标通常都不是要取得巨大增长，而是希望每天晚上都能吸引足够的顾客，让店主获得合理的收入。另一方面，尽管食品配送公司 SpoonRocket 从 2013 年起就一直在盈利，但它的目标是取得增长并持续不断地挣到更多的钱、扩张到新的城市并获取新的客户。这样看来，SpoonRocket 算得上是一家创业公司，但它会一直都是创业公司吗？是否会在某一时刻变成一家“成熟的公司”？要回答这个问题，我们不妨看看 Steve Blank 和 Bob Dorf 是怎么说的。

> 创业公司是一个暂时性的组织，目的在于寻找一种可重复、可扩展的商业模型。根据这一定义，创业公司既可以是一家新的企业，也可以是现有公司中的一个新部门或业务单元。
>
> ——Steve Blank、Bob Dorf，《创业者手册》

成熟企业拥有的产品已经被市场证明是为大家所接受的，所以它们关注的是扩大规模、优化产品和提升执行效率。而创业公司并不知道什么样的产品能在市场中立足，所以公司的主要注意力将放在试验、尝试和纠错上，重点是寻找一种可重复、可扩展的商业模型。可以这么说，创业公司的最后一个要素就是它们是按**探索模式**运作的。现在我们已经找到了所有的要素，把它们汇总一下，“科技创业公司”是具有下述特征的组织。

- 产品：技术。
- 环境：极度不确定。
- 目标：大幅增长。
- 运作模式：探索。

所以，我在书中并不关心一个组织成立的时间有多长，它有多少员工，所在的行业是什么，赚了多少钱。这本书的内容既可以用在全新的只有 3 个人的公司上，也可以用在有着 3000 人规模的成熟公司新成立的创新机构上。只要你进行的是技术研发，环境总处于变化之中，主要目标是为了增长，机构是以探索的模式在运行，那么书中的内容就适合你。这和多数人认为的“创业”可能有所不同，但我想不到有更好的词语或短语可以表达出这些意思。我曾经一度想过把这本书叫作 *Hello, Organization Designed for Massive Growth That is Searching for a Repeatable Business Model and Building Technology in an Extremely Uncertain Environment*（《你好，探寻可重复商业模型并在极度不确定的环境中研发技术以寻求巨大增长的组织》）——不过 *Hello, Startup* 听起来好像更性感一点儿，所以我就继续使用“创业”（Startup）这个词。

1.3　为什么应该在创业公司中工作

我们现在已经知道什么是科技创业了，但为什么所有人都这么关注科技创业呢？究竟是什么让它如此重要？要在科技创业公司中工作，甚至自己创立这样一家公司，我们应该考虑三个主要因素：更多的机会、更多的所有权以及更多的乐趣。

1.3.1　更多的机会

在此，我要告诉大家一个有趣的真相：我们其实是一个半机器人。一直以来，我们的大脑和身体都通过各种人造部件和技术得到增强。这种情况潜移默化地发生，让人无法觉察。但如果把你和所有附加的东西都送回数千年前，和那时纯粹的有机生命做个比较，你就会像有超能力一样。我们的身体机能之所以出现了明显的增强，正是因为有了现代机器的帮助，例如眼镜、隐形眼镜、助听器、填充物、支架、义齿、心脏起搏器、心脏瓣膜置换术、髋关节置换术、人造心脏、3D 打印耳朵、人工植发、隆胸、皮肤移植、钛合金骨头和假肢。不过，谈到技术对我们的影响，这一切也只是表面上的东西。

举个例子，这本书，或者放大了说——书写，就是一种增强思维能力的技术。我们可以把单词“存储”到纸上来扩展自己的记忆，也可以在白板上一步一步地解决数学问题来

扩展自己的计算能力，还可以给别人寄一封信、发送一封邮件或一条短信来扩展自己的沟通能力。每当我们画示意图、统计图、表格、时间线或蓝图时，都相当于通过书写增强自己的思考能力。

现如今，我们会经常通过数字媒介进行思考，你也许会在平板电脑或电子阅读器上阅读这本书的电子版，也可能通过在线书店（比如 O'Reilly、Amazon 和 iTunes）买到这本书。我们可以从 Twitter 或 Reddit 获取资讯，把自己的简历放到 LinkedIn 上，使用 TurboTax 进行纳税申报，在 YouTube 和 Netflix 上娱乐消遣，通过 Gmail 和 Facebook 和朋友保持互动。在我们的口袋里、手提袋中或桌子上，也许就放着一部手机。我们可以用它来增强自己的沟通能力（例如打电话、发短信）、记忆能力（例如日程提醒、闹钟、照片）、方向感（例如 GPS、Google 地图），获得更多的娱乐（例如音乐、视频）和更多的知识（例如 Google、Siri、Yelp、股票、天气）。手机已经成为了我们的一部分，你会随身携带，让它伴你入眠，每天不断把玩，总是离不开它。事实上，一旦没有了手机，我们很可能会感到失落和紧张。

如果我们现在出去走一走，旁边可能会有汽车、公交车和火车飞驰而过，这些都是技术带来的奇迹。它们都是在电脑上设计，在满是机器人的工厂里生产出来的，它们提高了我们在短时间内进行长途旅行的能力。现在抬头看看，可能正有飞机从头顶飞过，而驱动它穿过天际的正是喷气式引擎、无线电和自动领航等技术。在其之上，卫星和空间站正环绕地球轨道运行，它们拍摄照片、测量天气、处理电话呼叫路由。

但这仅仅只是开始。很快，我们会用上可穿戴智能设备（例如 Apple Watch、Google Glass 和 Jawbone Up），用手机来锁门（例如 August Smart Lock、Lockitron 和 Goji），用手机去监控和诊断疾病（例如直接通过手机跟踪血压和心电图情况以提前发现心脏疾病），依靠机器人而不是人去完成各种各样的任务（例如用 Roomba Vacuum 机器人代替清洁工、用 Amazon 的 Drone Delivery 无人机快递替代联邦快递），使用“复制器”创建物体（例如在家打印 DNA 或用邮件把扳手发送到外太空），乘坐机器人控制的交通工具出行（例如 Google 或特斯拉的无人驾驶汽车）或者进行太空旅行（例如通过 Virgin Galactic 或者 SpaceX 实现）。

那么，这些技术都有什么共同点呢？它们全都依赖于软件。换句话说，就像 Marc Andreessen 在 2011 年预测的那样——“软件正在蚕食世界”。因为科技愈加无所不在，软件公司将会占据越来越多的产业。例如，Amazon 在图书产业占据了统治地位，在新书销售和在线图书销售中分别占据了 41% 和 65% 的份额。在美国娱乐产业，现在有 50% 的家庭使用的是 Netflix、Hulu 或 Amazon Prime，而 YouTube 对 18~34 岁人群的覆盖率已经超过了任何一家有线电视网络。在旅游领域，在 Airbnb 登记的房屋已经超过了 100 万套，并以每周 20 000 套以上的速度增长。我们不妨把 Airbnb 和洲际酒店集团做个对比，后者是世界上最大的酒店公司之一（它们拥有假日酒店和洲际连锁酒店），也仅拥有 700 000 个房间。在通信产业，WhatsApp 的用户每年发送的消息达 7.2 万亿条，而全球的通信行业每年发送的短信是 7.5 万亿条，Skype 用户每年拨打超过 2000 亿分钟的国际电话，这个数量已经占据全球通信行业的 40%，增长率超过其 50%。软件公司在其他许多行业也逐渐彰显优势，比如招聘领域的 LinkedIn，支付领域的 Paypal、Square 和 Stripe，交通领域的 Uber 和 Lyft，音乐领域的 Spotify 和 Pandora，等等。

最大的变化还是来自移动领域。智能手机搭载了各种可以改变生活方式的软件并将一切都封装在一个盒子中，包括更快的 CPU，更多的内存及存储空间，前所未有的连通性

（3G、LTE、WiFi、蓝牙、NFC、GPS），大量的内置传感器（麦克风、摄像头、加速计、指纹识别、陀螺仪、气压计、距离传感器）、触摸屏和扬声器。这个盒子虽然小巧却用处很大，无论去到哪里都可以一直带在身边。所以说，移动领域成为了人类历史上发展速度最快的技术领域（见图 1-1）。

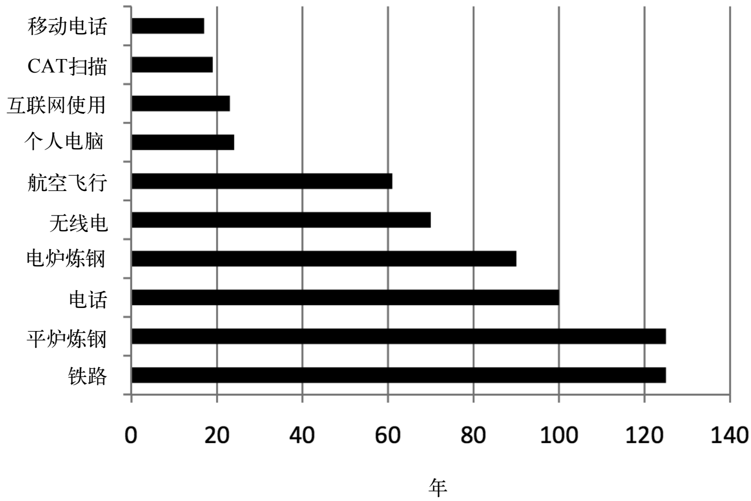


图 1-1：特定的技术达到 80% 覆盖率所需的年数（由 William Jack 和 Tavneet Suri 提供，数据来源：世界银行）

移动技术领域的相关数字令人吃惊。如图 1-2 所示，地球上使用手机的人远远比使用电视、银行账号，甚至比安全饮水和使用牙刷的人还多。到 2020 年，地球上将有 80% 的成年人使用手机。可以这么说，移动技术正在蚕食世界。

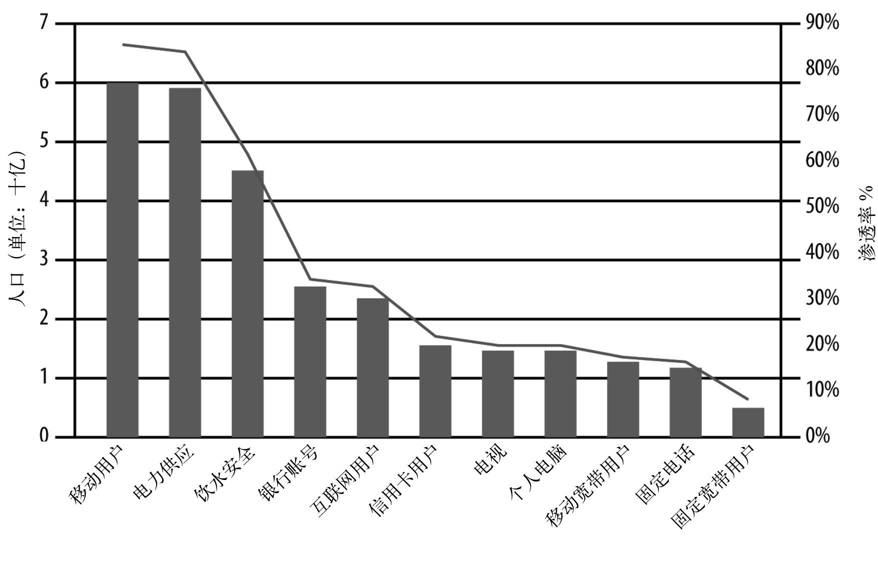


图 1-2：移动技术和其他技术在全球范围的应用比较（由 Chetan Sharma 提供）

软件和移动领域的大量变革正是由科技创业引领的。变革就意味着巨大的改变，对于创业公司而言，它们能够比大公司更好地应对（和发起）改变。一些科技巨头对此的应对方式，就是尝试像创业公司一样去运作它们的部分机构[1]，但很多都无法持续下去，最终将被创业公司所取代。事实上，每一代创业公司的增长速度，都要比之前的公司快出许多，如图 1-3 所示。Facebook、Google、Groupon 和 Zynga 这样的公司在 10 年内所取得的增长要快于整个 20 世纪的大多数公司的增长。在 1958 年，一家公司被纳入标普 500 指数的平均任期是 61 年；今天，这一数字已经下降到区区 18 年。

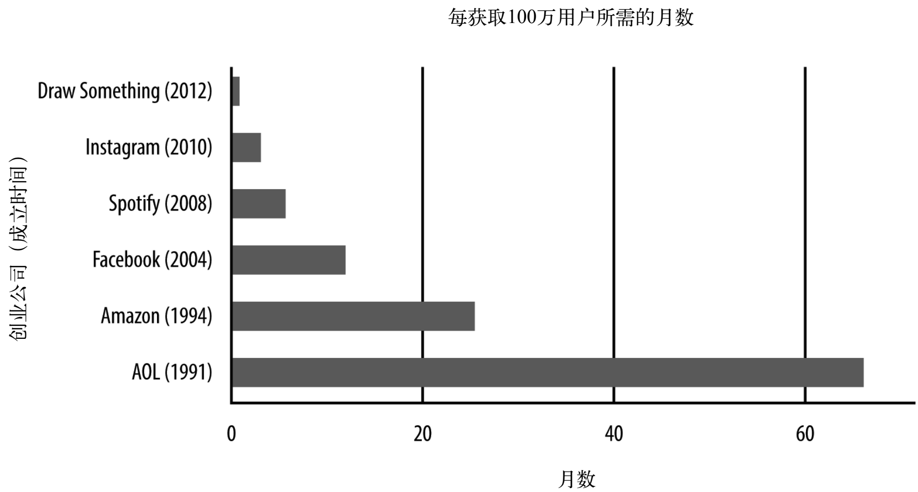


图 1-3：每获取 100 万用户所需的月数（数据来源：Fralic 2012）

如今，创业公司达到 10 亿美元估值的速度是 2000 年时的两倍，这并不是因为存在着泡沫，而是现在建立并发展一家公司，比以往任何时候都要容易。以下列出了创业障碍降低的一些因素。

开源

　　现如今，创业公司并不需要从头开始编写所有的东西，它们可以利用一千多万个开源代码库中的代码。其中许多代码库都是由大型社区中的开发人员开发并经过了测试，也有完善的文档。因此，使用开源代码不仅可以节省时间，而且相比较内部开发的项目而言，可供我们使用的开源项目规模更大、质量更高。第 5 章将更详细地介绍开源技术，教大家如何选择技术栈。

服务

　　通过利用数以百计的服务，创业公司的建立和运行变得分外简单快捷。例如，它们可以使用 AWS、DigitalOcean 或者 Rackspace，而不用去搭建自己的数据中心；也可以使用 New Relic、KISSMetrics 或者 MixPanel，而不用去研发自己的监控软件；可以使用 Amazon SES、MailChimp 或者 SendGrid，而不用去搭建自己的 email 服务；如果需要 logo，可以使用 DesignCrowd；如果需要法律服务，可以使用 RocketLawyer；如果

注 1：例如，Google X 是 Google 的一个半保密分支机构，该机构永远处于“探索模式”中，对可穿戴技术、无人驾驶汽车、高空 Wi-Fi 气球、葡萄糖监测隐形眼镜等项目进行研究。

需要接受付款，可以使用 Stripe；如果需要管理客户数据，可以使用 Salesforce；如果需要提供客户支持，可以使用 Zendesk。

分发

不管是推广自己的产品，还是运营一家雇员遍布世界各地的分布式公司，信息分发比以前任何时候都要更加简单。对于市场推广而言，技术已经无所不在，互联网和手机让我们可以通过搜索引擎、移动应用商店、广告、邮件和社交媒体渠道（比如 Twitter、Facebook、LinkedIn、Reddit、Hacker News 和 YouTube）把产品立即呈现在人们面前，触及的人数远远多于以往（阅读 4.2 节了解更多信息）。如果是建立一家分布式的公司，我们可以获得种类繁多的协作工具，比如 GitHub、Skype、Google Hangouts、JIRA、Slack、HipChat、Basecamp、Asana、Trello 以及其他一些产品。

信息

目前，如何建立成功创业公司的相关信息越来越多，来源包括图书（比如你现在看的这本）、课程（比如免费的斯坦福在线课程“怎样创业”，对本书就做了非常精彩的补充）、博客（特别是 Paul Graham 的文章）、聚会小组、会议、创业加速器和孵化器。

资金

有了上述开源代码、服务、简单易行的分发、更多的信息，创业比以往所需的资金更少。在确实需要资金的时候，我们也有很多办法，不仅可以找传统的风险投资公司，也可以寻求天使投资者（例如 AngelList）、众筹基金（例如 KickStart、Indiegogo、Lending Club 和 Kabbage）、政府基金和创业鼓励政策（例如，纽约的 Startup-Up NY 计划和新加坡”政府创业基金和援助方案”）的支持。

所有这一切都意味着我们正处于历史上一个不同寻常的时代。软件正在取代每一个产业，智能手机正在改变生活的方式，创业公司与以往相比，可以用更少的时间影响更多的人。可以这么说，软件正在蚕食世界。作为程序员，我们得到了这样一个史无前例的机会去加入这场盛宴，加入创业公司去编写代码，从而影响数百万人的生活。

1.3.2 更多的所有权

那么，我们为什么不在成熟的大公司中编写代码呢？在创业公司中工作究竟能得到什么好处，是微软、思科或 IBM 这样的科技巨头所无法提供的？在一家拥有数千名员工、已经存在许多年、工作也更有安全感的更加“稳定的”公司工作，不是更好吗？

好吧，我们来说说工作稳定性的问题。我们的父母或祖父母很可能会在同一家公司工作 50 年，顺着职业的阶梯不断攀爬，戴着金表退休[2]。现在，我们再也享受不到这种待遇了，因为那种工作已经消失了。据统计，美国 60 年代初期出生的大多数人会在 18~46 岁从事 11.3 份工作，这个数字可能在不断地上升，20 世纪 80 年代初期出生的大多数人到 26 岁时已平均从事了 6.2 份工作。从第一个数字可以算出，平均一份工作的持续时间还不到 3 年。大公司的工作并不比小公司的工作更稳定。例如，仅仅在 2014 年，思科就解雇了 6000 名员工，IBM 解雇了 13 000 名员工，微软解雇了 18 000 名员工，HP 则解雇了

注 2：这是过去一些美国公司的传统，会在工作超过三四十年的员工退休时送上一块金表，象征员工把时间献给了公司，公司又将时间赠还予员工。这一传统可追溯至 20 世纪 40 年代的百事公司。
——译者注

27 000 名员工。所以，所谓的工作稳定性已经不复存在。

> 我在大学毕业正决定去哪儿的时候得到了一条建议：你应该把硅谷当作一家大公司，其中有 Facebook 部门、Google 部门和一大堆小型创业部门。有时候部门会发生重组而不再独立存在，但所有的人只要加入其他的团队就可以了。我觉得这是一个非常好的比喻，在这里人们会相当频繁地在不同的公司间流动。
>
> 哪怕你不是一个能力出众的软件工程师，也真的不必担心加入创业公司有什么风险。你应该可以得到合理的薪水，也许不像在大公司的收入那么高，但足以支付自己的账单和贷款，也还过得去。如果那家创业公司倒掉了，再去找另一份工作就行，真的不是什么风险。
>
> ——Tracy Chou，Quora 和 Pinterest 软件工程师

真正的风险并不是因加入了小型创业公司而失业——毕竟我们在大公司工作也没办法保证不失业——而是失去机会的风险。如果选择了在一家公司工作，实际上也就是选择不在其他许多的公司工作。在这个意义上，缺乏工作稳定性也许并不是一件坏事。如果同一份工作已经干了很久，我们很可能正在错失其他一些更好的机会。

停滞不前是在大公司工作的一个普遍问题。到头来，你总是一遍又一遍地做着相同的任务，觉得工作不再有挑战性，便会停止学习，对工作感到厌烦。此外，在这样的公司工作，你对自己从事的事情也没有多少话语权，做出的贡献也经常被认为是无关紧要的。在大公司工作的你有点在像一艘有着上千名划桨手的大船上的一员，你就这样做着重复、辛劳的工作，但贡献却完全淹没在他人的船桨所泛起的波澜中。如果要评功劳，也只会评给掌舵的人，虽然他们除了戴着一顶显眼的帽子之外，好像也没做多少事情。虽然你对船前进的方向没有什么话语权，但当你确实想努力一把去影响它时，会发现让一艘大船改变方向几乎是不可能的[3]。

> 从我自身的经历看，在较大的公司工作，你的成功或失败通常都取决于所在的团体，如果上层管理者觉得该团体完成的工作是战略性的，和商业目标是一致的，他们就会关心你们什么时候可以带来收入等诸如此类的事情。虽然这样的工作看起来很重要，但我喜欢的却是能对自己的命运更有发言权的环境。在那种环境下，成功或失败取决于一个人的执行力以及他是否能做出市场需要的东西。
>
> ——Julia Grace，Weddinglovely 联合创始人、Tindie CTO

在小公司工作，通常会拥有更多的**自主权**，对自己做什么、什么时候做和如何去做都有更多的话语权，也没有那么多的繁文缛节、官僚主义和政治斗争。最重要的是，作为创业公司的创始人或早期员工，你可以参与定义公司的文化（阅读第 9 章了解更多信息），例如，公司的使命是什么？它的价值是什么？在沟通交流时是要开诚布公，还是深藏不露？你们打算用开放式隔间，还是私人办公室？允不允许员工在家办公？是通过一定的管理层级来组织公司，还是保持公司的扁平化？你们会对工作和假期进行严格管理，还是只关注结果？在大公司，这些决策大部分都已经是板上钉钉了，唯有忍受。而在创业公司，有许多决策都是取决于你自己。

注 3：显然，如果你在一艘满载的、以正常速度行驶的邮轮上，当你看见前方有冰山时，便为时已晚了。

我们在小创业公司所做的每一个决定对于公司都有较大的影响。此外，这种影响也会很快见效，因为通常而言，小公司与大公司相比有着更快的反馈回路。你所编写的每一行代码，实现的每一个功能，都会有看得见的差别。你也不再只是大机器中的一个小齿轮，而是对整个组织都有显著影响的人。你会感到和公司的使命联系得更加紧密，更能提升你的**使命感**。在大公司中，人们很难把提升公司的利润空间放在心上，但是在小创业公司，你要对它的生存负责，所以更容易受到鼓舞，更容易感到它是与你息息相关的。

创业也可以让你更加具有**掌控力**。在创业公司中，你会面对各种各样的任务，在自己前进的路上经常需要学习新的东西。可能某一天要编写数据库查询，第二天又要设计用户界面，之后还得回复客户的服务邮件，中间又要腾出时间准备投资者的融资演讲稿，期间培养出的这些技能将对你今后的职业生涯大有裨益。你也会学到如何应对紧张、压力和风险，会被推出自己的舒适区之外，这才是你真正能学到东西的地方。这也就是为什么很多人在创业公司三个月要比在大公司工作三年学到的还多。

> （在之前的公司）多年以来，我都觉得公司有许多需要花大力气去改变的东西，但我甚至无权和人们争论应该怎么去做，我的义务不过就是服从前人的决定，他们的想法就是我做事情的正确方式。
>
> 在 Foursquare，工程师屈指可数，多数的决策都还没有人定，我可以自己去做决策，这样感觉好很多。结果也确实如此，我做出了许许多多的决策，虽然未必都是好的决策，但是在三年半之后，我的感觉就是：抱歉抱歉，但因为这是我三年前做出的糟糕决定，我可不能撒手不管。这真的是一种很好的学习体验，无疑让我受益匪浅。
>
> ——Jorge Ortiz，LinkedIn、Foursquare 和 Stripe 软件工程师

综上所述，自主权、掌控力和使命感是激励人的三个最强有力的因素（阅读 9.5 节了解更多信息）。如果你找到了一份可以同时提供这三者的工作，那么就是找到了一份你会热爱的工作，也是一份你可以为之自豪的工作。

1.3.3 更多的乐趣

创业可以有更多的乐趣。在大公司，我们面对的是已经在市场中存在的产品，所以主要任务就是去优化它。在创业公司，我们面对的只是一堆猜测，得去想什么东西或许可以在市场中立足，重点是探索。你会发现，这样的探索可以让你收获更多的乐趣。

这种探索可以看作是你和世界的一场战斗。你要为存活而战斗，要与世界建立更紧密的联系，而不仅仅是去努力提高 2% 的利润空间；你也要努力为世界带来新的东西，这远比优化已有的东西更激动人心；你会为你的首次产品发布会、首次盈利或 IPO 而欢呼雀跃，这远比每年的圣诞晚会或通过最近的业绩考核更让人难忘。

> 说实话，虽然硅谷给了我丰厚的收入和所有的一切，但我生命中最美妙的一刻，却是我们的一位合伙人在半夜给我打的一通电话，那是我人生中收获的最大快乐。他说："有人给我们付了 50 块钱！"这是用户使用我们的软件所需要支付的费用。我们通过 PayPal 收款，而这笔钱刚存入我们的账户。我的脑子里就浮现出这样的情形：我们做出了软件，然后把它放到网上，现在有人真的为了它而向我们支付真金白银。我不太敢取这笔钱，因为……好吧，我其实是担心我

们的软件会出故障，那个客户又会回来把这 50 块钱要回去。我都不知道到时有没有 50 块钱还给他，所以还是别去碰那钱好了。

——Vikram Rangnekar，Voiceroute 和 socialwok 联合创始人

即便是创业时那些“狗血”的日子，也可以是充满乐趣的。贫民窟一样的办公室，必须勒紧裤腰带去维持生计，还常常会有不知道自己究竟在做什么的感觉，这一切都会让人感到害怕，但同样也是激动人心的。它们会教你学会感激生命中的小胜利，而不是去纠结于职位的升迁或权力斗争。

我在 LinkedIn 最美好的回忆来自最初加入的时光——在内河码头东部（East Embarcadero）的办公室里度过的前两年。虽然那时几乎没有什么收益，但我们仍然热爱着那里的工作。午餐通常就是冷冻的墨西哥卷饼或者随便从快餐车买的东西，还得看看那天快餐车有没有出现。这和现在硅谷创业公司里工程师们的奢华待遇形成了鲜明的对比。即便如此，我们那时仍然有很好的待遇。我最美好的记忆就是夏日的一天，Reid Hoffman 自己掏钱请一辆雪糕车停在办公室旁，让公司里的所有人都去享用。

那是一间不可思议的办公室，它坐落在垃圾场、机场、高尔夫球场和东帕罗奥图中间。卫生间总是“洪水泛滥”，我们还被偷了好几次。但那是个非常好玩的地方，我们在办公室玩滑板车比赛，搞吉他英雄竞赛，玩 Nerf 玩具枪大战。Ian McNish 会拿着他的巨型火箭筒玩具，跟在别人后面对着后脑勺就是一枪，差点把人弄成脑震荡。

还有一件事是我真的很喜欢的，那就是我们每周的全体会议，所有的产品经理和工程师都会来到会议室里仔细查看那些数字。我们大概在 2005 年秋天推出了人员招聘产品，之后的感觉就像是：等等，难道我们在赚钱了吗？人们真的会为这东西付钱？后来，我们就开始赚到了一百万美元。那感觉就像：“天啊，我们赚了这么多钱啊！”

——Nick Dellamaggiore，LinkedIn 和 Coursera 软件工程师

创业公司天生就和变化密不可分，所以它们对从事不同寻常的事情会更加开放，这也是为什么大多数有趣的公司文化都是创业公司建立起来的，而不是来自大公司。你可能已经听说过大多数科技公司必备的东西，比如比较宽松的着装规定和免费的零食、饮料和餐食，但其实远远不止这些。例如，HubSpot 会定期请思想领袖来举办演讲，无限量地为员工报销书费，每三个月会有一次半随机的“座位洗牌”，还有一项无条件的假期政策。Evernote 也有一项无条件的假期政策，但他们更胜一筹，会为实际休假的员工提供 1000 美元的奖金。在 Asana，员工可以获得 10 000 美元，自己决定添置什么办公室设备，公司还提供了内部瑜伽和按摩服务，还有一位全职大厨可以在现场为员工定做餐食（阅读第 9 章了解更多信息）。

这些东西听起来就像是些小恩小惠，但它们却可以让你改变做事的方式，抛弃“不过是另外一份工作”的想法。如果你足够幸运，可以顺路搭上“火箭宇宙飞船”——一个极其成功、超速发展的创业公司——你的人生也可能会为之改变。对我而言，在 LinkedIn 工作的日子就是一段不可思议的时光：把网站的规模发展到可以承担数亿的会员；在山景城、纽约、柏林、阿姆斯特丹和多伦多举办的黑客日竞赛；可以听到 Sheryl Sandberg、

Marc Andreessen、Ariana Huffington、Thomas Friedman、Cory Booker、Bryan Stevenson 甚至是奥巴马总统讲话的 InDay 系列演讲；还有纽约的 IPO；轮渡大厦、汽车运动俱乐部和巨人体育场的假日派对；纪念每一款产品发布的 T 恤和庆典[4]，不一而足。有时候，我都很难相信有人会为我支付这一切费用。

创业公司具有不走寻常路的勇气去尝试全新事物，正是这样才造就出一个个极佳的工作场所。如图 1-4 所示，具有不走寻常路的勇气也是让自己的生活非比寻常的关键所在。

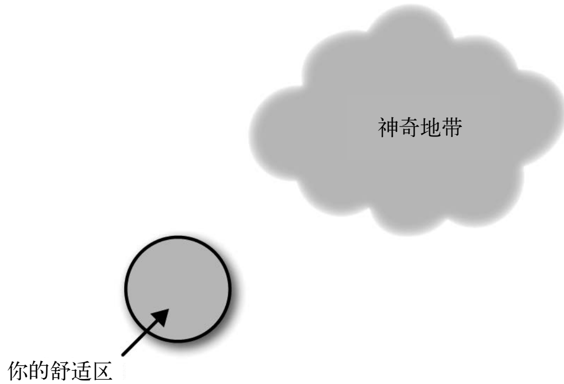


图 1-4：神奇地带

1.4 为什么不应该在创业公司工作

到目前为止，本章看似在讲创业公司无论在哪一方面都比成熟的公司好。情况也并非如此，创业公司自身也有许多问题，其中有一些问题比大公司更为严重。事实上，创业公司往往很极端：高的更高，低的更低。

并不是每个人都适合加入创业公司，而适合创业的人就更少了。在这一节，我会向大家展现创业领域存在的一些弊端：创业并不是那么光鲜亮丽，要让人做出许多牺牲，并且很有可能无法致富。我也会讨论怎样权衡考虑是自己创业还是加入别人的创业公司。

1.4.1 创业并不是那么光鲜亮丽

史蒂夫·乔布斯登上了《时代》杂志的封面，埃隆·马斯克登上了《财富》杂志封面，Twitter 也时常出现在电视荧幕上，还有一部关于 Facebook 的电影。科技创业者们已经成为了新的摇滚明星，一些程序员甚至还有自己的经纪人。在很大程度上，这是好事情。凡是能引起孩子对技术兴趣的事都是好事，创业者或程序员可以说是比摇滚明星或运动员更好的榜样。但是，和媒体经常出现的问题一样，这些“光鲜亮丽”会让人们对创业领域的真实情况产生扭曲的想象。

看见创业者出现在每本杂志的封面上，人们会产生一种错觉，认为创业英雄可以凭借一己之力想出绝妙的战略，克服所有障碍，打败所有竞争对手，从而改变世界，并在此过

注 4：我还过考虑过把这本书命名为 *How to Never Pay for a T-Shirt Again*（《怎样才能永远得到免费的 T 恤》）。

程中变得富有。像《社交网络》这样的电影把创业生活描绘成无穷无尽的聚会和胜利；而现实当中，根本没有哪个创业者或创业公司会像那样。首先，绝大多数创业公司都会失败，只有少数的创业公司可以成功，而成功的公司并不是因为他们有一个可以“顿悟”的英雄，而是因为他们有一个团队在日复一日地打磨雕琢，让产品和公司不断地迭代进化。每一个创业公司背后的真实情况无不包含了大量的失误、失败、权衡、争论和斗争。偶尔，也会有背叛或危机，还常常伴随着恐惧、压力和痛苦。最终，胜出者通常不是提前想出完美计划的杰出战略家，而是一个哪怕事事不顺也能生存下来的斗志旺盛的团队。

换句话说，创业是一种 99.9% 都是困难、一点都不浪漫的工作。星期四晚上 11 点，当你的亲人待在家里，身心放松地坐在电视机前，你却还在部署新的代码。星期五凌晨 2 点，你的朋友全都外出聚会，你却还在猛敲代码修复 bug，因为这是产品的发布前夜。而整个周六和周日，正常上班的人已经不用工作，可以去徒步或自驾旅行，你却不敢离你的电脑五步远，因为网站需要 7 × 24 小时不间断运行，这一周你要随时待命。

大公司就可以享有专人专岗的奢侈，但是在小型的创业公司，每个人都必须成为通才，每件事你都必须要做一点。你得会安装小隔间、得会估算卫生间的厕纸要多少钱、知道怎么去招聘副总裁或销售人员，知道怎么生成工资单、填写各种法律和税收表格、为投资者准备融资演讲稿、设计 logo，一大堆事情。有些程序员挺喜欢这样的工作，因为他们可以学到很多新技能，但也有些程序员更愿意只编写代码。

> 创办公司和解决有趣的技术问题之间存在巨大的差异。创业公司是有一些有趣的技术问题，但是一般情况下，公司成败并不取决于如何解决这些技术问题。当然也有例外，那就是关注解决困难的科学问题的创业公司。例如，我的一位朋友有一家电池公司，该公司在商业上是否成功既取决于他们在科学上的突破，也取决于他的公司运作是否良好。当然，99% 的 Web 创业公司都不存在这样复杂的科学问题，大部分的 Web 创业公司能否成功，几乎都完全取决于执行、有针对性的推广、销售、产品和技术。我们觉得只要像工程师那样，编写出色的代码，做出一些可以供百万人使用的东西，就可以获得成功，就可以获得业界的赞美，然后就会有人花几百万美元把我们招入麾下。那就是我们在 TechCrunch 和聚会上听到的故事，但现实情况远非如此。
>
> ——Julia Grace，Weddinglovely 联合创始人、Tindie 公司 CTO

在大公司工作的开发人员常常会觉得，创业公司的大部分工作和技术无关是一件很不可思议的事情。大公司本身也有一些让你分心的事情，使你无法安心编写代码，比如无效的会议、冗长的流程和规矩（请阅读 9.9 节了解更多信息）。在创业公司，虽然非技术的任务通常是工作中不可或缺的一部分。这种工作对于创办公司而言是必不可少的，但可能也是很无聊的。尽管创业公司名声在外，让人以为都是“迷人的”工作环境，但其实很多时间都被各种苦差事和明显不迷人的任务填满了。而且，只要你在组织中越往高处走，花在自己喜欢的技术任务上的时间就会越少。

> 我喜欢那种投入的状态，砰砰地敲着代码，猛然察觉到了午夜，发现自己刚刚居然做了这么酷的东西出来。随着你把它发展成为了公司，当上了领导，你会意识到自己不能再像以前那样了。这种情况是慢慢发生的，实际上你都不会注意到。

> 当公司只有 5 个人的时候，你几乎不会花很多时间去和人们谈职业发展，考虑他们的晋升周期和工资标准。当你的公司发展到 50 人的时候，你会发现这些事情要花掉你 10% 的时间。当你的公司发展到 100 人，底下有四五个人要向你汇报工作，他们每个人都领导着 20 个人，你会发现这些事情要占用掉你 50%~75% 的时间。当公司继续壮大，顷刻间，你又发现自己开始成为公司的公众形象，你编写代码的最后一点时间也会被挤占掉。如果你总是要四处奔走，和投资者会谈、做演讲、进行谈判，你会发现所有的剩余时间全部被抽走了。变化无常，你已经很难腾出大块的固定时间去做真正的技术工作了。这种变化的过程非常缓慢，直至有一天你醒过来才意识到："噢，天啊，我已经有 4 个月没有写代码了。"
>
> ——Steven Conine，Wayfair 创始人

许多开发人员在从编写代码的角色转变到领导角色（比如 CEO、CTO 或副总裁）的过程中都会遇到一些障碍。如果你从来没有担任过领导角色，也许会期待作为执行团队的一员，会觉得自己很重要、受尊敬和有影响力。你会想象自己把全部的时间都花在制定战略、发布命令、运筹帷幄上，活像一名五星上将。在现实当中，你可能更多的是介于销售人员和精神病医生之间。你要花大量时间试图随时随地让某个人或所有人去关注你的公司；你也要花大量时间去倾听员工的声音，努力解决他们的需求，处理他们的抱怨，想方设法去激励他们；你还要去做决定，但大部分的决定都是痛苦、有风险且不受欢迎的。而且不论你多么努力，有些决定都可能是错误的。有些人可以在这样的环境下茁壮成长，但如果你不是这样的人，可能就不是个当领导的料。

> 人们都希望自己创办公司当 CEO，从而站在金字塔的顶端。有些人会受此愿景的激励，但结果却未必如此。真实的情况是：其他所有人都是你的老板——你的所有员工、客户、合作伙伴、用户和媒体都是你的老板。现在我宁可没有那么多的老板和需要为之负责的人。大多数 CEO 的生活就是向除他以外的每个人做汇报，至少这是我自己和所认识的多数 CEO 的感受。如果你想对别人施加权力和权威，请从军或从政，别去当创业者。
>
> ——Phil Libin，Evernote 公司 CEO

1.4.2 创业就是牺牲

建立一家成功的创业公司的难度是超乎想象的。面对大公司的竞争，你很难招聘到出色的人才；当你好不容易才招聘到的出色人才决定离开公司时，也会让你感到世事维艰；解雇那些表现不好的人也很困难；激励人同样困难；在事情不见起色钱却已花完时，激励自己更是困难；融资是困难的；在投资者介入之后，不让他们使自己的业务偏离轨道也是困难的；在你不得不担心公司的短期生存问题时，关注公司的长期发展方向也是困难的；让不断变化的市场接受全新的产品是困难的；在某种东西上花那么多的时间——制作、销售、推广，仍然不引起别人的注意，坚持下去也是困难的；每一天，你都要在几乎没有足够信息的情况下做出数十个决定，而且每一个决定都会把很多的时间、金钱和许多人的职业置于风险当中，这同样不容易；你犯了错误，并且还会不断再犯的时候，同样也是艰难困苦的，因为除了自己之外，你没办法去责备别人。

上面所说的一切，表明在创业公司工作需要做出大量的牺牲。有些人可以比别人更好地应付这一切，但在新成立的创业公司工作，通常就意味着你没有足够的时间经常和自己的朋友及家人在一起，可能身体也会受不了。创业可能会毁掉你的婚姻，导致你的精神和身体出现问题，极端情况下，甚至有创始人精神崩溃而自杀。当然，情况很少会怎么糟糕，但长时间压力过大是普遍存在的问题。

> 我那年只有 26 岁，却来到了医生的诊室。我出现了短期的记忆问题。经过验血，医生对我说："你的数值就像是 60 岁的人，肯定是有问题了。"这时我才意识到自己不能再这样走下去，过后我就告诉老板："喂，我要离开了，我已经受不了了。我一周要工作 90 个小时，已经持续了八九个月了。"他回答我说："是的，我正在医生这里看心脏问题呢，我可能也要离开了。"由此我懂得了，在创业公司工作必须要控制好自己的步伐，需要努力工作，但也必须找到一种可持续的方式去做这些事情。
>
> ——Philip Jacob，Stylefeeder 创始人、Stackdriver 和 Google 软件工程师

创业就是坐着情绪的过山车，有时极高，有时极低。对有些人而言，这是创业魅力的一部分；对有些人来说，这种压力已经超出了他们能够掌控的范围。对创始人而言，压力尤其大。如果你是创业公司的员工，公司的失败只会让你感到沮丧，但你可以拍拍屁股去找下一份工作。但如果你是创始人，公司的失败会让你感到拖累了所有的人，员工为你付出了青春，客户给予你金钱和信任，投资者给了你投资，家庭给了你支持，最终，你一事无成。你的梦想已死，这才是毁灭性的。

1.4.3　你可能不会变得富有

大多数的创业都会失败，当然这个数字在很大程度上取决于如何定义"创业"和"失败"。但通常来说，失败率大概是在 75% 左右。因此，四次创业有三次是失败的，尽管存在着各种的痛苦和牺牲，但创业之路就是如此。就算你是少数能够成功的幸运儿，也仍然未必会因为创业而变得富有。这是因为在创业领域，回报的分布遵循幂律分布，即少数的赢家会获得大部分的金钱。在对 60 多万家创业公司自 2000 年开始的数据进行分析之后，我发现仅仅 34 家公司（Facebook、Twitter、LindedIn 和 Uber 这样的知名巨头）就占据了总市值的 76%。如果你是这些巨头之一，就可以变得富有，但属于这样一家公司的概率是相当低的。相反，如果你在其他的创业公司中，即便取得了成功，回报也是非常小的，并且大部分都到了投资者手中（阅读 10.3.2 节了解更多信息）。

你也不应该期待通过薪水而致富。多数的创业公司在初期的薪资水平要低于市场，所以甚至可以说，加入创业公司实际上要承担着赚更少钱的风险。如果那家公司取得成功并壮大起来，你的薪水通常也会跟着增长，但很少能够弥补之前在收入上的损失。另外，你也不要设想因为自己是早期员工，就可以获得晋升而进入高层（例如 CTO、副总裁），从而弥补之前的损失。这是因为在早期，你面对的是长时间的工作、快速变化的需求和紧张的最后期限，这一切使得你几乎不可能做出非常高质量的软件。随着公司不断发展，这个赶工拼凑出来的遗留系统已经开始无法胜任工作，所以公司需要招聘更多"经验丰富"的员工去"收拾乱摊子"。如果这个系统是靠你个人英雄式的努力才做出来的，那这正会成为招聘新员工的理由，对于你获得高层职位并没有强大的说服力。

简而言之，想通过加入创业公司而致富并不是明智之举。这不仅是不可能的事情，而且也是一种不好的导向。对金钱的渴望并不足以让人忍受在建立公司的过程中所经受的那种残酷的艰难工作。实际上，它甚至还可能会降低你的积极性，我将在 9.5 节讨论这一问题。

> 我要提醒你，经济上的成功并不是唯一的目标或者成功的唯一指标。人们很容易沉浸在赚钱带来的满足感中。你应该把金钱当作真正要做的事情的助推剂，而不要把金钱本身当作目标。金钱就像汽车中的汽油——你需要加以关注，否则车就会抛锚，但是美好的生活并不是在加油站间旅行。
>
> ——Tim O’Reilly，O’Reilly 媒体创始人

1.4.4 加入创业公司和自己创业的比较

这一章已经提到过，作为创始人和创业公司的早期员工，对创业的体验是完全不同的。以下就是这二者基本的权衡取舍：作为创始人，你必须做出 10 倍的牺牲以换取得到 10 倍回报的机会。我说的牺牲，是你将面对一个数量级以上的压力、风险和漫长的时间；所说的回报，也就是用这种痛苦所换取的回报。一旦成功，你可以赚到一个数量级以上的金钱和声誉。创立公司是高风险、高回报的游戏，而大多数人并不具备处理这么多风险及应对如此大压力的能力，所以大多数人都不应该去当创业者，不论他们有多么美妙的主意。

即便你可以应付这种压力，还有另一个需要考虑的因素。这是我在写这本书的时候偶然想到的，它完全改变了我对创立公司的思考方式。作为创始人，如果你幸运地打造了一家成功的创业公司（请记住，这个概率大概是四分之一），平均来说，你要花上七八年才能够成功地退出（例如被收购或者 IPO）[5]。当然，对于投资者来说就真的只有“退出”了，而创始人通常来说都会至少多待上几年[6]。所以，我们可以从中得到的一个经验法则就是：只有你愿意把生命的下一个十年都花在开创一家公司上，你才能去做这件事。

如果你现在 20 岁，你就要在公司一直工作到 30 岁。如果你现在 30 岁，那么在 40 岁之前做不了别的什么事了。我知道了这一统计数字，就去查了一下自己的创业点子清单，把上面一半的点子都划掉了。我认识到，其中有许多点子从本质上看都只不过是“快速致富”的方案，我不可能把下个十年花在它们身上去奔波劳累。

正如之前所说的，成功的退出也不是创办公司的唯一理由，它可以说是从事与创业相关的事情中最糟糕的一个目的——但是，有如此多的人把创业看作是一种快速致富的方法，如果本章的其他内容不能让你记住，请一定要记住这点：建立创业公司很可能不会让你变得富有；如果可以，也不会那么快。成功是极少的，如果确实发生了，也要花上近十年的时间。

注 5：我们可以看看过去十年间大多数成功的创业公司进行 IPO 或被收购的时间：Facebook 是 8 年，Google 是 6 年，Twitter 是 7 年，LinkedIn 是 8 年，WhatsApp 是 5 年，而 Zappos 是 10 年。

注 6：事实上，如果创始人试图在 IPO 之后立即离开公司，对他的声誉、公司和股票价格通常都会带来损害，所以大多数的创始人都至少会再多逗留几年。对于收购方而言，多数合同都包含了 1~2 年的“cliff”（最短生效期）或“vesting period”（等待期），以帮助公司完成过渡。经过这一期限后，创始人才能获得收购方的经济回报，所以这种合同通常也被称为“金手铐”。

在这十年期间，你不得不非常辛苦地工作。这种辛苦程度要超过你生命中从事的任何其他工作。现在开办一家公司可能比以前更为容易，但让它成功的难度却丝毫不减。所有的创始人都会告诉你，把新的产品带入市场，改变用户的习惯，招聘到合适的人，最终实现收支平衡——这些都属于生命中最难做到的事情。

> 我认为最困难的事情就是成功函数是非常不连续的。例如，你努力了几个月绞尽脑汁要找出提高用户增长率的方法，尝试引入一些新特性，觉得可以让指标曲线“向上和向右”移动，但实际却一点儿作用都没有。长期以来，一无所获。然后突然之间，巨大的成功就毫无征兆地到来了。
>
> 既然我们无法提前知道这些间断点会出现在什么地方，那么唯一合理的行为似乎就是真正努力去工作了。时间的长度是有限的，所以只能尽可能多地去做事情，让自己在倒下之前碰到下一个断点的机会尽可能地大。如果你在到达下一个断点之前就已经倒下了，你也知道自己已经努力过，不可能更快地实现目标了。
>
> ——Martin Kleppmann，Go Test It 和 Rapportive 联合创始人

因为成功函数非常不连续，在创业公司工作，特别是作为创始人，就有点儿像戴着眼罩跑马拉松。你知道那是一场长跑，但你无法看到里程标记或时钟，对自己跑了多长自然毫无感觉，甚至无法确定自己是否跑在正确的方向上——但你不能降低速度或停下休息，否则肯定会有人超过你。所以只能尽可能快地继续前行，追逐下一个断点。

对大多数的程序员来说，加入别人的创业公司可以得到足够的好处，还可以极力避开那些缺点。事实上，作为程序员，把赌注压在几家创业公司身上，是找到成功职业的最佳方式之一。如果自己创立一家公司，它成为下一个 Google 或 Facebook 的概率是非常低的，但作为创始人，你必须是满怀坚定，只有为之坚持 5~10 年才知道结果如何。在同样的时间内，如果是员工，就可以加入三四家创业公司，每家公司待上几年，这样找到成功创业公司的机会将显著增加。

> Facebook 的第一百名工程师所赚到的钱，比硅谷 99% 的创业者所赚到的钱要多得多。巨大的馅饼就算切成小片，本身仍然是巨大的。
>
> ——Dusting Moskovitz，Facebook 和 Asana 联合创始人

就像现实中会有追逐救护车的律师[7]，在硅谷也会有追赶 IPO 和收购的工程师，这也不是什么坏事。这些工程师会在公司 IPO 过后跳到另一家公司，如此反复，他们实现产品，帮助扩大组织的规模，从而贡献自己的价值。作为回报，他们也得以培养各种各样的技能，享受每一家公司的独特文化，积累股票期权。在若干年后，他们会带着许多有趣的经历离开，很多时候，口袋中的收获也不会少。

如果你知道该关注哪些工程师，就可能会预测出哪些公司很快就会进行 IPO 或被收购。例如，在过去的几年，我观察到一些朋友会在 LinkedIn、Facebook 和 Twitter 间转来转去，会在每家公司 IPO 之前跳槽进去待上几年。他们又是如何知道的呢？其实有三个主要的信号。第一，选择你认识的人中大部分人已经在使用的产品。大多数的开发人员是

注 7：在美国，车祸发生时，有些律师会立即赶到现场，唆使受伤者提出诉讼，要求赔偿，自己从中渔利，这些律师被称为“追赶救护车的人”，泛指那些急于拉生意，不顾职业道德的律师。——译者注

“早期采用者”，如果他们中的许多人都蜂拥而上地用某一种技术，那么其他人很有可能很快就会追随而来。第二，寻找那些已经通过多轮融资而筹集到许多资金的公司。这些公司获得的投资金额越多，投资者就越希望能得到巨大的回报，而实现巨大回报最常见的方式就是让公司上市或者被收购。第三，寻找那些以超乎寻常的速度增长，并且在盈利之前需要更多金钱去维持增长的公司[8]。

如果加入别人的创业公司更有可能致富和得到乐趣，创立自己的公司还会是个好主意吗？是的，只有在你不能不做的时候。也就是说，创立公司最合适的原因就是你对某个想法充满热情，非要把它带给全世界。你之所以去做，不是为了声望或财富，而是因为它对你足够重要。为了实现它，你愿意为之经受所有的痛苦、风险和牺牲。

也一定不要把完成特定使命的梦想（阅读 9.2.1 节了解更多信息）和创业的梦想混淆起来。有时候，创业是实现梦想的最佳方式，但是很多情况下，最好还是从事居家业务（例如在家工作的顾问）或者加入别人的公司，或者在大学里面做研究。创业只不过是达到目的的手段。

1.5 小结

> 你知道什么是创业中最美好的事情吗？……你只能体会到两种情绪：狂喜和恐惧。并且，我发现睡眠的缺乏会放大这两种情绪。
>
> ——Marc Andreessen，Netscape、Loudcloud、Opsware 和 Ning 联合创始人

现在我们已经了解到创业生活的正反两面。创业可能会带给你更多乐趣，也可能会让你更有压力。你会得到更多的自主权，也要做更多艰苦的工作。你可以对自己的职业和全世界都带来巨大的影响，也非常有可能会失败。关键的问题是：你是否适合创业？

这个问题的答案只有一个：尝试。这不意味着所有人都应该走出去开办公司，但一辈子至少也要尝试一次，每个人也都应该体验一下创业公司的工作。说到这一点，其实每个人也应该体验一下在大型、成熟的公司工作。创业公司未必适合所有的人，大公司也是如此。所以不妨二者都试一试，看看哪一个更适合你。

> 我在大公司和小公司都工作过。我觉得二者都值得体验，因为它们分别需要不同的技能。创业公司有一种活力感，你所做的是能够引起人们共鸣的新东西，可以改变他们沟通交流、旅行或做任何事情的方式。在大公司工作时，需要有沟通以及感悟他人观点的能力。但有时，如果我们只不过想要完成某件事情，但最后银行账户上却来了三百万美元，并且也没有人妨碍你，那种感觉会很不错的。
>
> ——Philip Jacob，Stylefeeder 创始人、StackDriver 和 Google 软件工程师

也许尝试过后，你会发现创业生活就是你想要的，甚至也可能会受到鼓舞而成为一名创业者。在某种程度上，每个人都已经是创业者。亚当·斯密写道，每个人“在一定程度

注 8：对 2015 年和 2016 年的公司进行观察，至少根据他们的资金募集情况、增长情况和近期开发人员的迁移模式，符合上述特征的公司有 Uber、Airbnb、Square、Stripe、DropBox、Pinterest、PagerDuty、Slack、Zenefits 和 GitHub。

上都是商人”，你是在把自己的时间、知识和资源出售给他人，无论是他人的公司还是自己公司的客户。多年从事一份工作、慢慢晋升的日子已经一去不复返了。自我雇用（self-employment）的人数规模空前，点对点经济不断增长，这一切都是由 Uber、Sidecar、Lyft、Airbnb、TaskRabbit、Homejoy 和 Etsy 这样的创业公司促成的。

当然，出租一间房子或从事顾问工作和建立创业公司是不能相提并论的，但是随着自我雇佣的模式变得越来越普遍，人们会更加接受创业公司，对大公司“职业安全感”的错误依恋也会更少一点。人们甚至可以接受一种时髦的“工作”概念：工作并不是固定的实体，也不是只有在大学毕业后才有权做的事。没有所谓的工作，其实你在做的只不过是他人（老板或客户）觉得有足够价值的事，这样他们才会付给你钱。

> 在你成长的过程中，人们总是会告诉你：这个世界就是……尽量不要撞了墙也不回头，要努力拥有美好的家庭，要学会享乐，要存下一点钱。
>
> 那是一种非常有限的生活。生活可以变得更加多彩，只要你发现这样一个简单的事实：你周围的一切，即你所谓的生活，都是由不如你聪明的人组成的，你可以去改变它，可以去影响它，也可以做出自己的东西供他人使用。
>
> 一旦意识到这一点，你将从此不同。
>
> ——史蒂夫·乔布斯

第 2 章

创业点子

所有创业公司都是从一个想法开始的。Google 的创办源于他们认为网页间的超链接类似于学术论文间的引用，能够以相同的方式进行排名。LinkedIn[1] 的创办源于他们认为专业人士在互联网上发现其他专业人士的最佳做法就是通过他们所信任的人脉。DropBox[2] 的创办源于他们认为在计算机之间共享文件，肯定有比使用 USB 记忆棒更好的方法。

当准备写这本书的时候，我问一些朋友更想知道创业哪方面的内容，他们的答案是：创业者们是如何想出绝妙的创业点子的？不少人都认为史蒂夫·乔布斯、里德·霍夫曼、亨利·福特和拉里·佩奇都拥有创造力这种“超能力”。他们认为创造力和大多数超能力一样，具有二元对立的属性——要么拥有，要么没有。

在这一章，我希望能够让大家相信，创造力也是一种可以学习的技能。和所有技能一样，可能有些人在这方面会更擅长一些，但任何人都可以想出好点子。想知道怎么做到吗？本章的前半部分会探讨点子究竟从何而来，后半部分则将描述如何验证点子是否值得转化为产品。

2.1 点子从何而来

对点子最大的一个错误认识就是认为它们是自然而然从脑子里蹦出来的，一蹴而就，凭空产生。一想到好点子是怎么产生的，你可能就会想到托马斯·爱迪生在工作间里第一次让灯泡亮起来，想到苹果掉在艾萨克·牛顿的脑袋上，或者会想到阿基米德把自己沉到浴缸时喊出的那句“尤里卡”[3]。虽然这种顿悟的时刻在讲故事时既省事又让人印象深刻,但

注 1：LinkedIn：全球最大的职业社交网站，是一家面向商业客户的社交网络，中文名为领英。成立于 2002 年 12 月，2011 年 5 月 20 日在美上市，总部位于美国加利福尼亚州山景城。——编者注

注 2：DropBox：成立于 2007 年，提供免费和收费服务，能够将存储在本地的文件自动同步到云端服务器保存。在不同操作系统下有客户端软件，并且有网页客户端。——编者注

注 3：原文为 Eureka，即希腊语“我发现了”之意，据说是古希腊学者阿基米德因在泡澡时悟出根据比重原理测出希罗王王冠所含黄金的纯度而兴奋不已时发出的惊叹语。——译者注

实际上大多数点子都不是这么诞生的。

事实上，阿基米德从未在自己的著作中提到过“尤里卡”这个词——这个故事来源于维特鲁威，一位生活在阿基米德之后近 200 年的罗马作家，大多数科学家都怀疑这个故事是维特鲁威虚构出来的。同样，苹果实际上也从来没有砸在牛顿的头上，他并不是在某一时刻突然发现了地球引力原理，而是经过了 20 年的研究。而爱迪生也并没有发明电灯泡（在爱迪生开始研究电灯泡之前，它已经存在 70 多年了），而是发明了灯丝使电灯泡成功地商业化。而他也不是在一瞬间就发明出来的，而是使用不同的灯丝材料进行了 6000 多次试验。

点子并不是灵光一现，而是有一个发展和进化的过程，这将是贯穿全书的一个主题。点子也不会凭空发展进化出来，物理学的能量守恒定律阐述了能量从来不会凭空产生或湮灭，而是以不同的形式被重新利用，点子也同样遵守这样的守恒定律，所有新的想法只不过是现有想法组合而成的结果。我们可以把自己脑海中的信息想作是一个个离散的数据点，一个点子只不过是这些数据点之间的连接。要产生新的点子，并不是凭空地生成一些新的数据点，而是将已有的数据点连接起来。我们不要把新点子当作是头顶上突然亮起的灯泡，更好的比喻应当是把新的点子想作是点亮灯泡，让它照亮已经存在的东西。

你见过的所有新出现的、有创造性的东西，都只不过是集之前的点子之大成。这个概念在 Kirby Ferguson 拍摄的名为 *Everything is a Remix* 的系列视频短片中有很好的体现。例如，微软的 Windows 操作系统从 Apple 公司的 Macintosh 模仿了许多特性，而后者早期的大多数点子又是从 Xerox PARC 的 Alto 计算机那里借鉴的，而 Alto 计算机本身主要也是受到斯坦福研究院的 NLS 计算机的灵感启发。过去 40 年几乎所有的畅销歌曲全部都基于完全相同的四个和弦，从甲壳虫乐队的“Let it Be”、Journey 的“Don’t Stop Believing”，到 Bob Marley 的“No Woman No Cry”、The Red Hot Chili Peppers 的“Under the Bridge”、Lady Gaga 的“Poker Face”，概莫能外（如果不相信，不妨看看 Axis of Awesome 在一场趣味十足、让人大开眼界的表演中演唱的“The 4 Chords Song”去核实一下）。过去 10 年排名前 100 的电影中有 74 部是续集、重新制作或是对图书、卡通、漫画书的改编。现在有 11 部《星际迷航》、12 部《13 号星期五》、12 部《007》。甚至还有像《变形金刚：月黑之时》这样的电影，它是一部电影的续集，而原来的电影又源于一部动画片，动画片则起源于孩之宝公司（Hasbro）的玩具，这一玩具又起源于日本的玩具。你是不是被我绕晕了？

在本章开头提到的所有绝妙的创业点子也都是这样重新合成的。Google 并不是第一个搜索引擎（在它之前至少有 10 个搜索引擎，比如 Yahoo、Excite 和 AltaVista），它最重要的点子——基于引用分析和文献计量学领域的网页排名（PageRank）算法——至少在 60 年代早期就已经存在了。LinkedIn 并不是第一个社交网络（它在很大程度上受到了 SixDegrees.com 的启发，甚至取得了 Six Degrees 的专利），而且也不是第一家针对专业人士的线上人脉网站（Ryze、Xing 和 Spoke 都在 21 世纪早期就已经创立），更不是第一家线上职位公告板（Monster 和 HotJobs 在 90 年代后期就已经创立），甚至不是 Reid Hoffman 在社交空间领域的第一次尝试（早在 1997 年他就创办了一家名为 SocialNet 的约会社交网站）。

甚至可以说你现在正在阅读的这本书也不过是我将数以百计的参考文献和我对其他创业程序员的访谈进行了重新整合。这一节更是如此，因为它的内容大部分借鉴 *Everything is a Remix* 系列视频和《好点子都是偷来的》一书。本书就像是对重新合成这一概念的一种

自我引用和元聚合。

这种混合和模仿看似是坏事，其实并非如此。现代社会有一种把模仿妖魔化为剽窃、欺骗和造假的倾向，并且通过专利和版权保护等手段来阻止这样的行为。但真实的情况是“我们都在用相同的材料做东西”（弗格森语），混搭和重新合成是产生新点子的常见方法[4]。那是因为创造力的产生可以归结为三个阶段，这三个阶段都不过是不同形式的重新合成：

(1) 模仿；
(2) 转换；
(3) 合并。

当我们学习一种新的创造性活动时，模仿总是要做的第一件事。婴儿通过模仿成人来学习，艺术家通过模仿大师来学习，程序员则通过复制粘贴来学习。转换类似于模仿，但对原有的点子做了一些改进，就像爱迪生为电灯泡研发新的灯丝一样。合并意味着要把一些已有的点子组合成比各部分都要好的一个整体。例如，古登堡并没有发明螺旋压力机、活字、墨水和纸张，但他能够把这些东西凑在一起做成印刷机，得出一个和每个组成部分都大不相同的新东西。

模仿、转换与合并已经深深地根植于每一个生物体当中。事实上，这也是我们之所以成为生物体的过程：我们的细胞进行自我模仿（有丝分裂）、转换（由随机突变引起）和合并（当你被繁殖而成的时候）。从非常实际的意义上看，你就是父母及所有祖先的混合。当然，这也不意味着盲目地偷窃别人的成果就可以想出新的点子，而是说创新的最佳方式是研究、注明出处、重新合成、聚合和转换。

所以，如果我们想要找到创业的好点子，就需要一整堆的“原料”，从而可以在此基础上去研究、注明出处、重新合成、聚合和转换，也就是说，我们需要掌握大量的知识。

2.1.1 知识

要想出许多新点子的最好方法就是学习大量老点子。既然新的点子只不过是把老的点子连接起来，那么我们脑海中的点子越多，就越能在这些点子之间建立联系。

> 知识和创造的成果就像利滚利。假设有两个能力差不多的人，其中一人只比另一人多做 10%，二者在产出上的差距将超过两倍。
>
> —— Richard Hamming，You and Your Research 演讲

如果知识就像利滚利，那么越早投资越好。从今天开始，想尽一切方法开始学习（阅读本书就是一个好的开端），哪怕只是一小点，其效果也会超出你的预期。但是，我们应该花时间去学习什么呢？最有效的一个策略就是努力成为一名 **T 型人**。

> 既是通才（在许多有价值的方面高度熟练—— T 的横），也是专才（在某个特定的学科中属于领域内最出色的—— T 的竖）。
>
> —— *The Valve Handbook for New Employees*

注 4：实际上，专利和版权的诞生是为了鼓励新想法的传播，但专利法和版权法已经发生变化，超出了公认的“知识产权”的法律范畴，现在更多的是在扼杀创新而不是帮助创新。

我们倒过来讲，先介绍专才，再介绍通才。

1. 专才

要成为专才，就必须对某一个特定主题有强烈的、近乎着迷般的求知欲。这一主题不需要和创业或赚钱有什么关系——事实上，没有关系可能还更好，只要是某种以其内在品质就可以让你神魂颠倒的东西即可。这个主题应该是很具体的，可能比你在大学所学的专业还要具体。例如，如果你学的是计算机科学，你可以成为机器学习、分布式系统或计算机图形方面的专家，甚至也可以超出你的专业范畴，比如遗传学、认知心理学或用户界面设计。当然，你也没必要上大学去专门学习这样的主题，而是可以顺道就把这样的专业技能培养出来（例如把制作无人机作为爱好，从而成为机器人方面的专家），或者在工作中去培养（例如在银行工作，从而成为支付系统方面的专家）。

只有培养了足够的专业技能，才能进入所选学科的前沿领域。在这一不断探索的过程中，你会想出一些创业点子，例如 Larry Page 在图形理论、文献计量学和 Web 方面拥有足够的专业技能，从而推动了搜索技术的发展。Reid Hoffman 之所以打造了 LinkedIn，是因为他作为创业者和投资者，必须要成为建立人际关系网的专家，而在这一过程中，他认识到存在着通过互联网为专业人士建立人际关系网的机会。

要培养专业技能，需要进行大量的研究，包括阅读该领域内所有顶尖的图书和论文，研究相关行业中所有顶级的公司和产品，订阅相关的杂志、博客和出版物，参加会议和聚会，与领域内的专家取得联系（或者至少在 Twitter 上关注他们）。而且，还要进行大量的亲身实践，如果所选择的专业领域是全职工作的一部分，正好就是题中之意。这也说明了在开始自己创业之前，最好先在别人的公司工作一段时间。如果这不属于日常工作的一部分，就需要在业余项目、20% 项目和黑客马拉松上花点时间了（阅读 9.5.1 节了解更多信息）。

2. 通才

当今世上最有价值的创业都是组合多学科各种知识的结果。比如把对人类肢体的理解和专业技术知识结合起来，就得到了当今最热门的行业之一：生物科技。有数十家公司正在设计可以装在智能手机上的医学传感器，比如 AliveCor 的心电图阅读器、IBGStar 的血糖计和 FotoFinder 研发的用于皮肤癌筛查的电子皮镜。

要成为一名通才，你必须定期搜寻新点子。有些人天生就对所有东西都有好奇心，会觉得这很容易做到。如果你不是这样的人，也许需要刻意努力跳出自己的舒适区，体验各种各样的文学著作、电影、旅行和活动。有一种实现的途径，就是写出一些“top 5”清单。例如，你可以做一张所有文学体裁的清单（如历史、心理学、科幻小说、数学、计算机科学、生物学，等等），试着阅读每一种体裁中最出名的 5 本书；或者列出学校所有科目（例如数学、物理、历史、生物、英语，等等），每一门科目都去上五门主题课程，如果没有时间的话，可以阅读这些主题最好的教科书。

这是一种让自己大范围接触新点子的有趣方法。每当我这么做时，都会震惊于人类知识看似不相关的领域竟有这么多重叠的地方。我发现《写作法宝》中的写作建议竟然和《代码大全》中编写整洁代码的建议有那么明显的相似之处。我从心理学图书《思考，快与慢》中学到了产品定价的宝贵知识，这些知识和我阅读所有商业或经济类图书所得到的收获一样多。我甚至发现将我女朋友关于 20 世纪 40 年代东欧共产主义崛起的论文研

究用在解释当前硅谷的创业发展方式上也能有所洞察。

之所以存在这样的重叠，是因为大多数的图书、电影和课程实际上都是与人息息相关的。程序设计的图书并不是关于代码的，而是在讲如何编写人们可以理解的代码；心理学图书并不是关于大脑的，而是在讲人们是如何思考的；而科幻、奇幻和恐怖电影并不是关于科技、外星人或者怪兽的，而是在讲人们如何才能在非同寻常的处境下生存下来。所有的这些知识从根本上来说都是关于人的，所以相同的核心原理会以不同的形式不断地出现。由此也可以看出，这些原理在几乎所有的学科中都是适用的。就好比几乎所有的运动员都不单单要练习自己所从事的运动项目，还要在健身房进行力量和体能训练，因为这样可以锻炼出所有运动都适用的身体素质。同样，你不仅要研究自己的学科，还要接触更大范围的其他学科，培养出全面的思想素质，帮助你对任何一个专业领域都有更深入的理解。其目标，正如史蒂夫·乔布斯所说的，就是努力"让自己感受人类最美好的东西"。读者可以阅读第 12 章了解更多有关学习的内容。

只要你能很好地把知识的深度和广度结合起来，就可以把知识转变为点子。

2.1.2 点子的产生

> 单词或语言是书写和交谈用的，在我的思考机制中似乎不发挥任何作用。[5]
>
> ——爱因斯坦

你有没有想过你的点子是如何产生的？实际上这很难想清楚，因为创新思维好像就这么出现了。意识中出现的新点子就像会变魔术，就像有人在你潜意识的幕布背后把它递给你，就像一种无法控制的思维过程。然而，这么说并非完全准确。实际上，绝大多数思维过程都是无法控制的。

Tor Nørretranders 在 *The User Illusion* 一书中告诉我们，人类的大部分思维都是在潜意识下发生的。很明显，很多基本的身体机能都由我们无意识地控制，比如心跳、消化和荷尔蒙水平，但是潜意识的作用还远远不止于此。已经有许多研究在试图测量意识的"带宽"（即人可以处理多少信息），得到的结果通常都是在每秒 10~40 位之内。而有研究已经测量出，潜意识是以接近每秒 1100 万位的数量从感觉器官接受信息的。换句话说，"我们眼睛所见到的、耳朵所听到的，以及其他感官感觉到的东西，只有百万分之一会出现在我们的意识中"。因为你的潜意识会决定要丢弃哪些信息，哪些是要出现在意识中，这也就意味着"意识并不能引发行动，但它可以决定应该执行的行动"。换句话说，我们无法强迫自己的大脑进行创造性的思维，但如果恰好有这样的思维，我们可以对它们进行评估。这是不是说我们就无法影响到创造力呢？

也不尽然。我们无法强行控制自己的潜意识，但可以训练和引导它。这听起来可能有点奇怪，但其实我们一直以来都在这么做。每当我们学习一些新东西的时候，都是一种有意识活动的开始，经过足够的练习之后，就会成为一种下意识的活动。例如，当你开始学习驾驶的时候，要集中十分的注意力才能让汽车不偏离路线、在限速范围内行驶、记得使用转向灯。练习几年之后，所有这些动作都会变得"自动"，你已经能一边轻松地开车，一边听着收音机与旁人交谈。学习骑车或者学习基本的算术、阅读也是同样的道理。

注 5：*An Essay on the Psychology of Invention in the Mathematical Field*，Hadamard、Jacques 著，2007。

有意让自己的意识关注在某个特定的任务上，我们就可以逐渐地训练自己的潜意识去做这件事，最终，就可以完全依靠潜意识来进行了。事实上，我们对用潜意识阅读的训练已经炉火纯青，你**不用**它甚至都不行了。只要你看着这一页的内容，你的潜意识就在自动对它们进行处理，显意识的思维就会听到它们。

很明显，我们可以教会自己的潜意识骑车和阅读文字，但是怎么样才能教会它想出新的点子呢？要实现这一目标，我们必须让自己的潜意识置于可以培养出创造力的合适环境中。

2.1.3 培养创造力的环境

纵观人类历史，出现了不少**多重发现**（multiple discovery）的例子，即有两个或多个科学家或发明家在差不多相同的时间内提出相同的想法，比如牛顿和莱布尼茨都在 17 世纪发表了关于微积分最早的论文，达尔文和华莱士都在 19 世纪提出了进化论，而格雷和贝尔在同一天提交了电话的发明专利申请。这一切都不是偶然，它表明环境对新点子的涌现有巨大影响。

> 我并没有发明什么新东西。只不过把他人的发现汇集起来了而已，在他们背后是几个世纪的工作成果。如果我在 50 年、10 年，甚至 5 年前做这些事，可能就不会成功。任何新生事物都是如此。当万事俱备，质变就发生了，而且是不可避免地发生。教别人相信“人类最伟大的进步是由极少数人推动的”实在是糟糕透顶的胡说八道。
>
> —— 亨利 • 福特

那么，什么样的环境可以激励人们产生新的点子呢？因人而异，但最常见的要素有这么一些：

- 给自己充足的时间；
- 记录点子日记；
- 解决问题；
- 放下工作；
- 添加约束；
- 寻找痛点；
- 与他人交谈。

1. 给自己充足的时间

最影响创造力的因素之一就是时间。之前已经说过，点子并不是在顿悟的瞬间出现的，它需要发展和进化，有时需要很长一段时间，比如牛顿关于重力的想法，就花了 20 年的时间才逐步形成。创造力不能强求而得，也不能拔苗助长。如果试图这样做，譬如施加外部压力或悬赏刺激，实际上是在降低创造力（阅读 9.5 节了解更多信息）。

所以说，最要紧的就是要给自己充足的时间。在解决问题时，孵化期是必不可少的，所以在计划安排任何涉及创造力的活动时，务必多留出时间让潜意识去处理问题。

2. 记录点子日记

达·芬奇、居里夫人、爱迪生、理查德·布兰森[6]，几乎所有富有创造力的人最普遍使用的技巧，就是把自己的想法记在**点子日记**中。点子日记和普通的日记不一样，它不是用来记录每天做什么事情的，而是简短地记下一些备注、目标、意见、想法、问题、草图和观察到的东西。在一整天中，你可以随时在上面记东西。可以用一个小笔记本（比如一个 Moleskine 笔记本）和一支钢笔，也可以用手机上的录音机（如果你的手机可以用语音控制就会特别高效，按下一个按钮，说完“Siri，记下笔记……”就可以接着说下去了），还可以用移动笔记记录应用（例如 Evernote 和 Google Docs），email（比如说发一封主题为“#thoughts”的邮件给自己并增加一条过滤规则，把带有该标签的邮件自动移到一个专门的文件夹中），或者其他一些可以随身携带又方便使用的东西。

最重要的就是要使用方便。我们的目的只是把所有感兴趣的东西写下来，如果这一过程比较麻烦（例如得走到办公室，然后再打开电脑），点子日记就不是那么有效了。

其实，最大的阻碍可能还是你自己的判断。在这一阶段。先不要对自己的点子下结论。如果这些点子看似愚蠢、不完善或者让人尴尬，也没有关系，先写下来再说。并不是说写下来就一定要做，也没必要把它们拿给别人看，所以写下来并没有什么损失，反倒会有很多收获。因为写下一个点子就有点儿像种下一粒种子，慢慢地，随着时间的推移，如果还有一点运气，它也许可以成长起来；就算不会，它也是静静地待在你的点子日记里，不会打扰到任何人。

想要拥有好的点子，最重要的一个因素就是要先有很多很多的点子。当然，这里隐含的意思是，如果想要拥有更多好的点子，你也要有更多不好的点子。这个观点是有研究支持的，麻省理工学院和卡耐基梅隆大学的研究发现，产生不同寻常的点子的最佳方式并不是提高点子的**平均**质量，而是提高它们的**差异性**。也就是说，我们需要产生一些非常疯狂的点子，其中有一些糟糕得不可思议，但也有一些出色得难以置信。事实上，加州大学戴维斯分校的研究发现，科学及其他学科中的杰出成就者一般只是做得更多，而不是做的质量更高。

这就是为什么最为重要的是把每一个点子写下来，并且不要提前下结论——我们要习惯尽可能地多产生新的点子，特别是一些疯狂的、出乎意料的点子，不要因为自我怀疑而丢弃点子，阻碍了自己的进步。

> 有人说他要一两个月才能想出来一个好点子，问我是怎么想到的，因为我的好点子看起来好像比较多。我会问他，你每个月会想出多少糟糕的点子呢，他停了一下，说“没有”。
>
> 可以看到，这就是问题所在了。
>
> ——Seth Godin，Squidoo.com 创始人、作家

把脑袋里蹦出来的半成型的想法变成纸上具体的文字，可以让想法变得更加清晰，通常还会让你产生新的想法——也继续写下来。另外，仅仅把点子写下来这一行为就可以帮助你更好地记住点子——我已经丢掉了很多点子，就是因为确信自己可以记得住，但几

注 6：英国著名企业维珍集团的创办人兼董事长，对近 200 家公司有所投资，是当今世界上最富传奇色彩和个人魅力的亿万富翁之一。——译者注

分钟过后，就置之脑后了。

我们也要不时回过头去检查自己的点子日记。有些点子看起来傻傻的，这也没什么，跳过去接着往下读就行了。有些点子可能会让你觉得陌生，几乎就像是别人写的，感觉以前没见过。但也有少数的点子，几乎总是能够触发你产生一些新的想法，提醒你想起最近发现的一些新信息，或者会让你意识到，可以把它们和日记上另外的某个点子结合起来。点子就是以这样的方式慢慢地“成长”和“进化”的。如果说把点子写下来是在播撒种子，那么回顾和更新点子就有点儿像在浇水，甚至是在为植物交叉授粉。每当我回头看自己的点子日记，都会惊讶地发现情况已经发生了变化——好吧，也许是**我自己**发生了变化——每次都会给我带来新的体会。

3. 解决问题

由于点子来源于潜意识，我们需要让自己无意识地思考感兴趣的主题，因此最好的方式就是花大量时间有意识地去思考这一主题。

> 如果你深深沉浸在某一个主题中，日复一日致力于此，你的潜意识除了解决问题就不干别的了。也许在某天清晨或者午后醒来，你就找到了答案。对于那些并没有想方设法、全心投入解决现有问题的人，潜意识就会在其他事情上“游手好闲”，不可能有什么大作为。所以自我管理的方法就是一旦你有什么真正重要的问题，就不要把其他事情置于自己注意力的中心，你要一直把心思放在这个问题上。让你的潜意识保持在“饥饿”状态，不得不解决你的问题。这样你就可以平静地入睡，等待清晨醒来时得到答案，得来全不费工夫。
>
> ——Richard Hamming，You and Your Research 演讲

想办法解决问题，尽可能多地了解与之有关的内容，放下工作，让自己的潜意识有机会做自己的事情。

4. 放下工作

在点子日记上写下来的东西不应只有点子，还应该有你是在什么地点、什么时候有了这个想法的，还有那时你在干什么。如果你这样做了，可能会注意到：最好的点子不是在办公桌前工作的时候得到的。麦吉尔大学的研究发现，分子生物学家大部分最出色的点子都是在远离实验室的时候想到的[7]；爱因斯坦有一些最伟大的发现是在自己拉小提琴的间歇得到的[8]；许多人是在洗澡的时候想出他们最好的点子的；我则是在外面散步的时候可以产生最好的想法[9]。

所以，集中精神紧张地做一件事情，做了一段时间后，放下工作，找一些相当放松的事情做，让潜意识继续去解决问题，这样似乎是得到好点子的最好方法。之所以这样做，有几个原因。一是有规律地停下来休息可以帮助你打破在单一方向上的思维“定势”，你可以后退一步，看看更宏观的场景。另外，过度集中精力强迫自己工作对想出新点子是适得其反的。大脑只有在放松和额叶（用于分析思考和做决定的部分）大部分不活跃的时候才最有创造力。

注 7：《伟大创意的诞生》，Steven 著，2011。

注 8：*Nurtured by Love: The Classic Approach to Talent Education*，Suzuki, Shinichi, Waltraud Suzuki, 1993.

注 9：研究表明，走走路可以显著提高人的创造力。

要养成习惯，每天至少花 20 分钟做一些可以自我放松的事情，倾听自己的想法。可以是走路、洗个时间长点的澡、冥想、在吊床上躺一会儿、写写日记、画画、雕刻、做做木工或者放放音乐。不管怎么样，把点子日记放在边上，随时准备记下笔记。

5. 添加约束

读者们现在可以试一下《让创意更有黏性》一书中的一个有趣练习：设置一个 15 秒的计时器，尽可能多地写下你能想到的所有白色的东西。在往下读之前先做这个练习，我等着你。

你写下来几样东西了？

现在，我们进行这个练习的第二部分，把你的计时器重新设置为 15 秒，但这一次，请尽可能多地写下你能想到的冰箱中的白色的东西。

这次你写下来几样东西了？

大部分人都会发现他们在第二次练习中想到的东西和第一次一样多（或者多一些）。很明显，冰箱中的白色物品的数量应该是全宇宙中白色物品数量的一个微小的子集，但你可能已经发现冰箱的练习会更轻松一些。这是因为约束滋生了创造性。

另一个这方面的好例子来自于 *Not Quite What I Was Planning: Six-Word Memoirs by Writers Famous and Obscure*。这本书的开头是有关海明威的一个传说，讲他如何接受挑战，仅用 6 个单词就写出了一个故事。

> 出售：童鞋，全新。[10]
>
> —— 海明威

这本书还有其他一些以回忆录的形式展现的精彩故事，展示了仅仅用六个单词带来的非凡的创造性。

> "生于沙漠，却还口渴。"[11]—— Georgene Nunn
> "被癌症诅咒，被朋友祝福。"[12]—— 9 岁大的癌症幸存者
> "好吧，我本来以为挺有趣的。"[13]—— Stephen Colbert
>
> —— *Not Quite What I Was Planning: Six-Word Memoirs by Writers Famous and Obscure*

减少选项会提高自己想出创造性解决方案的能力，这种结论似乎有点违反直觉，但如果你认识到工作记忆要比长期记忆少得多，这就很好理解了。也许脑海中有数千个想法和概念，但你一次只能考虑一小部分。有点像玩杂耍，没有约束条件的系统就像把 100 个球都抛在空中，你是没办法全部顾得上的，你只会不断地把它们弄丢，又花大部分的时间去捡起来，重新开始。

如果你遇到麻烦，根本想不出什么点子，或者觉得自己拿了一张白纸，不知如何下笔，

注 10：原文为 For Sale: baby shoes, never worn。——编者注

注 11：原文为 Born in the desert, still thirsty。——编者注

注 12：原文为 Cursed with cancer. Blessed with friends。——编者注

注 13：原文为 Well, I thought it was funny。——编者注

约束条件对你会特别有帮助。有一种添加约束条件的方法是"活在未来，实现缺失的东西"。在一个名为 *The Future Doesn't Have to Be Incremental* 的演讲中，Alan Kay 描述了他们在施乐 PARC 的时候是如何"生活在未来的"。施乐 PARC 是一家引领前沿科技的公司，开创了现代个人电脑、图形用户界面、以太网、激光打印和面向对象编程等许多新领域。在 PARC，研究人员会玩一个游戏，后来被叫作"韦恩·格雷茨基游戏"，这是以史上最伟大的冰球运动员命名的。他把他的许多成功归功于一条简单的策略：我滑去冰球要去的位置，而不是它现在的位置。

"韦恩·格雷茨基游戏"也是类似的理念：你要用打冰球的这一策略，可能就要往后看 30 年或更长的时间，这离现在如此遥远，但你完全无须担心怎么走到那一步。你只要顺着"如果 30 年后，我们没有……那不是很可笑吗？"这样的思路来问自己就可以了。例如，当 Alan Kay 在 1968 年玩这个游戏的时候，它认为如果人们在 90 年代中期的时候还没有笔记本电脑和平板电脑，那就太可笑了。在 60 年代，这看起来像个疯狂的想法，当时大多数的计算机都是装着开关的大盒子（根本没有屏幕，更别提触摸屏了，也没有键盘和鼠标），但这个游戏的关键是不必去担心实现的细节，它就是要让大胆的点子进入脑海中。只要你找到喜欢的点子，就可以倒过来工作，看看需要怎么做才能把这一想法变成现实。

另一种添加约束的方法是走向另一个方向：寻找市场上刚刚出现的新技术，努力去寻找**现有的**实现方式与现在有了这些技术之后**可能的**实现方式之间的差距。或者，借用 Reid Hoffman 的提法，你可以问自己："世界应该是这样的吗？"Plangrid 公司就是一个很好的例子，他们注意到许多建筑公司仍在使用纸质的设计图，无论是创作、打印、分享还是更新，成本都非常高。随着平板电脑、无线互联网等技术的出现，Plangrid 意识到世界不应该是现有的样子，因此他们创建了可以在移动设备上数字化管理设计图的软件。

也许添加约束条件最容易的方式就是寻找一些存在错误或痛苦的地方。寻找痛点就是这么强大的一种手段，值得我们更深入地了解。

6. 寻找痛点

使用点子日记的最佳方式，是不仅记下点子，而且还要随时记下所有让你感到烦恼、使你痛苦、阻碍你前行或者仅仅是你觉得有问题的东西。每当你想说"这真是愚蠢，肯定有更好的方式"时，就把它写下来。换句话说，不仅要用点子日记记住解决的方案，也要把问题记下来。任何特别让你痛苦，或者频繁出现、影响了许多人的问题，都是潜在的创业点子。痛苦在哪里，机会就在哪里。

重要的是把问题写下来，即便你还不知道要如何解决。如果你一次次地看到问题出现，也许出现的场合略有差异，但每一次都可以简单地把想法记下来，然后你对这个问题的理解就会慢慢加深。最终，你也许会触及问题的核心，解决方案也就跃然纸上了。有时要过一段时间才能意外地发现解决方案，有时可能要很久以后才会有结果，有时是在完全不相干的情况下找到的，但除非你记住了最初遇到的问题，否则是不会觉察到那是个解决方案的。著名投资者、通用公司前研究主管 Charles Kettering 曾经说过："说清楚问题等于解决了问题的一半。"

学会鉴别和解决特别难解决的问题是一种宝贵的技能。Paul Graham 在他的一篇文章中说道："我们的周围存在着各种难办之事（schlep）"。"schlep"是一个犹太语单词，用来描

述一些特别乏味、让人讨厌的任务；人们通常会因为实在太讨厌这些事，而选择对它们视而不见。所以说，如果你乐意发现这些难办之事并准备好卷起袖子帮别人解决，你就可能会做出一些非常有价值的产品。

> 我了解的厌恶性盲区（schlep blindness）最显著的例子就是 Stripe，更确切地说应该是 Stripe 的点子。这十多年来，所有处理在线支付问题的程序员都知道这种体验有多么痛苦，肯定也有成千上万的人对这个问题有所了解。然而，当人们开始创业的时候，却打算去做菜谱网站或者本地活动的聚合。为什么呢？为什么他们要去解决这些很少人关注、没有人会为此掏钱的问题呢？什么时候才会有人去解决涉及世界基础结构的一些最重要的问题呢？正是厌恶性盲区使人们根本就没有产生解决支付问题的想法。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

7. 与他人交谈

和别人谈论自己的点子、问题或痛点是实现创新的一种强有力手段。这和把自己的想法写下来是很相似的。把自己的想法口头表达出来，让别人可以理解，这样也可以帮助你更好地理解自己的想法，有时候还会让你产生新的点子。其实也不是非得有另一个人在旁边，我有时候会出去走一走，大声地自言自语一番，甚至假装对着一群听众说话，虽然看似有点儿疯疯癫癫的，但这是一个相当有效的法子，可以产生许多点子（我会在走路的过程中把点子草草地记在点子日记上）。在编程领域，甚至专门有一个词用来描述这种技术，叫作“小黄鸭调试法”（rubber duck debugging）。当你面对一个非常棘手的 bug 时，你可以把详细情况说给一只小黄鸭听，或者讲述给其他任何一种无生命的物体。当你全部说完的时候，经常就可以找到解决方案了。

如果你旁边有别的人，他们的反馈、提问和纠正都会让你受益匪浅，因为通常来说，没有哪两个人头脑中的信息是一模一样的。一个人可能知道 A 和 B，另一个人可能知道 C 和 D，如果把 A 和 C 或者 B 和 D，或者 A-B-C-D 组合起来，一个新的点子可能就出现了。

> 我们在脑海中会不可避免地把分子生物这样的科学想象成：一个独自在实验室中的科学家弯腰驼背地扶在显微镜前，偶然撞上了一个重大的新发现。但 Kevin Dunbar 教授的研究表明，孤立的“尤里卡时刻”是很罕见的。相反，大多数重要的想法都是在定期召开的实验室会议上出现的。在这样的会议上，十多个研究人员聚集在一起，非正式地提出和讨论他们的最新工作。如果你看过 Dunbar 教授制作的点子形成示意图，就知道创新的引爆点并不是在显微镜里，而是在会议桌上。
>
> ——Steven Johnson，《伟大创意的诞生》

即便别人并不是你所讨论主题的专家，你仍然可以收获不少有价值的东西。我在写这本有关创业的书时，我的女朋友正在忙着她的东欧历史博士学位。这两个主题可谓风马牛不相及，但我们在写作的时候却发现了它们之间值得关注的一些联系，并发现定期讨论我们的想法和相互争论是一件很有帮助的事。因为她并不是创业方面的专家，所以我每次都要把我的想法解释给她听，期间会用到许多隐喻和类比，这会激发我脑海中大量的创造性思维。由于学历史的人和程序员在思考问题的方式上差异巨大，她给我的回应通常会让我站在不曾考虑过的新视角去想问题。

你可以在喝咖啡、喝冷饮、吃午餐、出去散步、会议期间，或者在正式的“头脑风暴会议”上和别人讨论你的想法。你可以尝试不同的方式，特别是如果你在记录点子日记的时候，不仅能记下自己的想法是什么，而且还能记下你是在哪里想到的，你就可以很快找到最适合自己的方式。你也可以和互联网上的陌生人交流你的想法，也可以把你的想法在博客、社交媒体（Twitter、Facebook 或 LinkedIn）和 Reddit、Hacker News 这样的讨论板上发布出来。你甚至还可以把你的想法摆在潜在客户面前，这部分内容将在 2.2 节详细讨论。

在详细讨论之前，先来看看拒绝和别人讨论自己想法的一个最常见的理由：秘密模式。

2.1.4 秘密模式

有一些创业公司在亮相之前，会尝试以**秘密模式**（stealth mode）进行运作，完全对外隐藏它们的点子和产品。这样做主要有以下两个原因：

- 害怕别人嘲笑自己的点子；
- 害怕别人窃取自己的点子。

第一个拒绝的理由源于害怕批评。大多数人在学校里接受的教育都是要求在每次考试中找到“正确”的答案，如果做不到，就会扣分。这种方式把大多数人教育得害怕犯错误，令他们不敢分享不完美的事物，也害怕失败。在创业领域，这和你需要具备的心态是截然相反的。Steve Blank 说道：“如果你在创业公司中害怕失败，就注定会失败。”犯错是学习应有的过程——在某些情况下，甚至是学习的唯一途径。毫无疑问，获得反馈是改进产品的最佳方法。

我认识的大部分成功人士都会刻意地定期去寻求反馈。他们会将处于未完成阶段的早期作品发给一群信任的朋友，然后在项目中吸纳点点滴滴的反馈，一点一点地让作品变得更好。他们最终得到的结果正是集众多思想之大成，远比单靠一人之力做出来的东西要更加出色。为此，我们不要把批评当作是对自己人格的攻击，那是别人在花时间帮你改进。我们一定要学会不要把人家说的“这很愚蠢”曲解为“你很愚蠢”。我们必须意识到，每一篇出色的文章都源自粗糙的草稿，每一本好书都需要一位编辑，每一名出色的运动员都需要一位教练。

> 隐藏自己的想法，会失去讨论带来的好处。探讨一个点子会产生更多的点子。所以最佳的方案是，如果你能够掌控，就找几个你信得过的朋友，开诚布公地交谈。这不仅是形成点子的方法，也是选择朋友的良方。能一动不动听你讲“异端邪说”的人，肯定也是你最应该去了解的人。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

第二个拒绝的理由是认为有人会窃取你的点子。通常来说，这并不是一种合理的担心，因为多数人对于“窃取创业点子、把它们带走、变成自己公司的东西”这类事情并不感兴趣——因为他们太懒，或是太忙了。更重要的是，他们并不像你一样对这些点子有很深的印象或那么大的热情。Howard H. Aiken 说过：“别担心人们偷走你的点子，如果你觉得自己的点子非常棒，还得让人们接受才行。”如果你不相信，不妨去 http://www.hello-startup.net/resources/startup-ideas 看看，了解一下别人的创业点子，看看有多少是你想“偷”去做成公司的。

实际上，真正吸引竞争者的并不是点子，而是点子**受到广泛关注**。只有在你发布了产品，并且已经开始显露峥嵘的时候，别人才会想着去抄袭你，所以不用担心在早期讨论你的点子。另外，如果你担心有人偷听你的点子而想把它偷走并打败你，这样的点子很可能没有防御性，无法实施。虽然这里用了**防御性**这个词，但我想表达的是，一个出色的商业点子应该具备某种**差异性**，可以让你和竞争对手之间产生巨大的差距（阅读 3.2.2 节了解更多信息）。例如，比较一下“我有一个照片分享应用的新点子”和“我有一个低成本发射物体到太空中的点子”。如果你的点子天生并不具备防御性，就算你能够在发布之前做到一直保密，竞争者仍然可以在你发布之后轻而易举地抄袭你的点子，然后打败你。而且，大多数行业实际上并没有所谓的“先发优势”。这个词最早在 1988 年由一篇名为 *First-mover advantages*（《先发优势》）的论文提出，但是 10 年之后，同样的作者发表了另一篇名为 *Fisrt-mover (dis)advantages*（《先发劣势》）的论文。在后者中，他们放弃了原有的许多主张。此外，1993 年的一项对 50 种产品中的 500 个品牌的研究发现，几乎有一半的市场先行者都失败了，而幸存者的平均市场份额要远低于其他同类数据。

仅仅有点子还成不了业务，理解这一点同样重要。业务是由点子和执行力构成的。也许有人可以偷走你的点子，但偷走你的执行力就要困难得多。我们来更深入地看看点子和执行力之间的关系。

2.1.5 点子和执行力

硅谷有一个流行的文化基因——点子是毫无价值的，执行力才是一切。这是一种错误的二分法。创业不单单是一个点子领头，跟着一些所谓的完全不用动脑筋、重复的“执行”过程。创业是成千上万次地重复“发现问题 – 想出点子解决问题 – 执行这些点子”的过程。

(1) 问题：我们需要一间办公室。
(2) 点子：我们在山景城（Mountain View）租个办公场地。
(3) 执行：在 Google 上搜索可用的办公场地。

(1) 问题：山景城的办公场地超级昂贵。
(2) 点子：我们要从投资者那里筹集一些钱。
(3) 执行：通过 LinkedIn 联系人搜索投资者。

(1) 问题：我不认识任何投资者。
(2) 点子：我们要找可以帮我们引荐的人。
(3) 执行：在 LinkedIn 联系人上搜索认识投资者的人。

可能是因为有一些所谓的“商业领袖”向人们展示了一个模糊的点子，就说它是通往财富之路的船票，于是就出现了“点子毫无价值”这样针锋相对的口号。但是，正如同点子不是诞生在某个“尤里卡时刻”一样，创业既不是某一个单独点子的结果，也不是某一次单独执行的结果。想要获得成功，需要持续不断地有好点子和高效的执行力，把它们割裂开单独拿出来讨论是毫无意义的。

> 这不是一个点子，而是一个点子迷宫：你需要沿途做出一千个决定，一些会导致死亡，一些则不会。最终的产品是穿过迷宫的一条成功路径，但它并不展现所有可能遭遇的失败。
>
> ——Balaji S. Srinivasan，斯坦福创业项目工程课程

想出一个点子之后，穿越“点子迷宫”的第一步就是检验一下市场是不是也同你一样，认为它有价值。这一步骤叫作验证。

2.2 验证

住在硅谷最好玩的一件事就是随时随地会有人在你身上试验他们的点子。当创业公司发布它的食物配送服务时，旧金山通常都是第一个可以使用的城市。当 Google 试验它的无人驾驶汽车时，会选择在山景城推出这些汽车。当我乘坐捷运列车时，会遇到完全不认识的人向我推销他们的创业点子。这一点 Aaron Levie 说得最好。

> 住在湾区，基本上就表示你接受成为未来各种疯狂生活方式的 beta 测试者。
>
> ——Aaron Levie，BOX CEO

这些人都想做同一件事情：在市场上**验证**他们的想法。也就是说，他们想测试自己的想法是否可行，是否已经找出了一个真实的问题，是否可以据此建立起有价值的商业模式。不管你自认为你的想法有多么出色，经过了多少深思熟虑，你本人是多么聪明，所有人都无法预言一个点子是否会成功。哪怕是风险投资者，他们的全部工作其实也就是挑选出赢家，投入巨大的资源去验证那些点子（例如雇用经验丰富的创业者作为合作伙伴、通过广泛的关系网对创始人进行审核、执行尽职调查），但他们的投资仍然有超过 60% 是亏钱的[14]。如果连他们都分辨不出哪些点子会成功，哪些不会，那么就可以说没有人能做到了。

> 有一个令人吃惊的事实，不管大公司还是小公司，是成熟的企业巨头还是全新的创业公司，在尝试推出新产品时，10 次中有 9 次都是失败的。
>
> ——Steve Blank，《四步创业法》

创业成功之路充满了失败。实际上，失败并不是一个合适的词。正确看待创业和产品的方法是把它们看作试验。试验的目的是要支持或者反证一个假设。对于科学家来说，试验无所谓失败，只会学到东西。就像爱迪生在尝试了用数以千计的材料作为灯丝但都失败之后说的：“我并没有失败，我已经发现了一万种行不通的方法。”

最有价值的学习通常都来自于意料之外的试验，就像 Spencer Silver 博士想尝试研发一种超强的黏合剂，但却反而得到了一种黏力非常弱，但可以重复使用的材料。Silver 博士并没有把它看作是失败，而是坚持做了下去，几年之后把它变成了一种产品，成了他公司最出名一种产品：3M 便利贴。

所有的成功公司身后都留下了一长串不成功的试验：

- Google：Wave、Buzz、Labs、Health、Video、Answers、Notebook、Audio Ads；
- Facebook：Beacon、Places、Credits、Deals、Questions、Gifts、Lite、Email；
- LinkedIn：Answers、Events、Twtiiter 和 GitHub 集成、Signal；
- Virgin：可乐、衣服、伏特加、婚礼、Vie 化妆品、汽车、Pulse、葡萄酒、牛仔裤；
- Apple：Apple Ⅲ、Lisa、Macintosh Portable、Newton、eMate、G4 Cube、Ping。

有时候，如果某种产品行不通，整个公司都不得不改变方向，这也称为**转型**。

注 14：Fred Wilson 写道：“早期风投很像棒球，如果每三次你能够击中一次，你就可以进入名人堂了。”

- Instagram 刚创立时名叫 Burbn，是一个类似 FourSquare 的本地分享移动应用。照片的分享仅仅是该应用的一个功能。但是随着照相越来越流行，公司转型发布了一款名为 Instagram 的新移动应用。几年之后，他们被 Facebook 以十亿美元收购。
- Groupon 起源于一个名为 The Point 的政治行动网站。为了勉强维持生计，CEO Andrew Mason 把 Groupon 作为一个副产品推向市场。当 Groupon 开始起飞的时候，整个公司进行了转型，聚焦在 Groupon 上。几年之后，Groupon 已经价值 120 亿美元，并且成功上市。
- Twitter 最初是一个叫 Odeo 的播客平台，公司在苦苦挣扎中决定转型到微博领域。Twitter 在 2013 年公开上市，估值大约为 180 亿美元。

不成功的产品和转型并不意味着失败，它们就像文章的草稿，在你完成之前总要修改几次，不仅写作是如此，生活中任何不简单的事情都是如此。在你想出好点子之前，没有任何捷径可以避开最初大量糟糕的点子给你带来的障碍。当然，在创业领域，你的时间和金钱都很有限——这通常称为创业**跑道**——要想顺利起飞，你需要尽可能快地分辨出哪些是糟糕的想法。这一概念通常称为**快速失败**，但正如前面所说的，我觉得这里用“失败”这个词并不合适，这会让我们觉得失败变成了目标，那么鲁莽和草率也就没什么问题了。实际上，尽快学习才是目标，所以我更喜欢用**速度制胜**这个词。

2.2.1 速度制胜

我在 TripAdvisor 工作的前几天就差点儿就迟到了。我本来是要去和 CEO Stephen Kaufer 见面，参加新入职人员午餐聚会。我四处乱撞，结果好几次都错过了他的办公室。最后，一位秘书为我指明了方向：“找到那扇挂着一张纸的门就是了。”我朝着她指的方向跑过去，终于看到了，他办公室的门上有一张 A4 纸，上面潦草地写着两个词。一走近，才发现写的是：

> Speed Wins（速度制胜）

这就是 TripAdvisor 的准则，Kaufer 在那天午餐的时候解释了这个准则，后来也多次在公司的全体大会上做过解释。创业成功，无论在哪个层面上，归根到底都取决于速度。你必须更快地实现产品、更快地编写代码、更快地招聘，最重要的是，必须更快地学习。

为什么总是要如此仓促？慢慢花时间把事情“做对”不是更好吗？要回答这个问题，我们先看看图 2-1。在理想的世界中，我们有了一个点子，会努力进行概念上的验证，向用户进行展示，发现用户喜欢之后投入更多的精力扩大概念的验证，最终得到一个稳定的、成功的产品。

只要你做过产品，就知道现实中的情况永远不是这样的。如图 2-2 所示，在现实世界中，你有了一个点子，会投入一些时间进行概念验证，展示给用户看，发现它不可行，然后又会回到绘图板上，想出另一个点子，进行另一次概念验证，再向用户展示，又会发现是不可行的。这样的过程要一次次地重复，直到——如果运气好的话，最终可以找到实际可行和值得扩大规模投入的产品。

这意味着试错阶段就要花掉大部分的时间，如图 2-3 所示。在试错中，最快发现错误的人将会胜出。

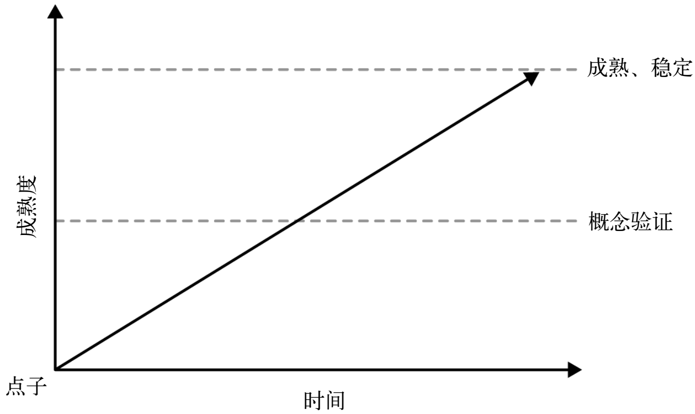


图 2-1：产品开发（理想情况）

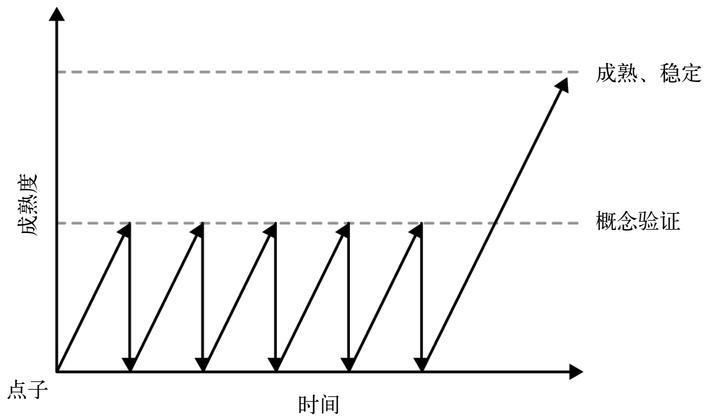


图 2-2：产品开发（实际情况）

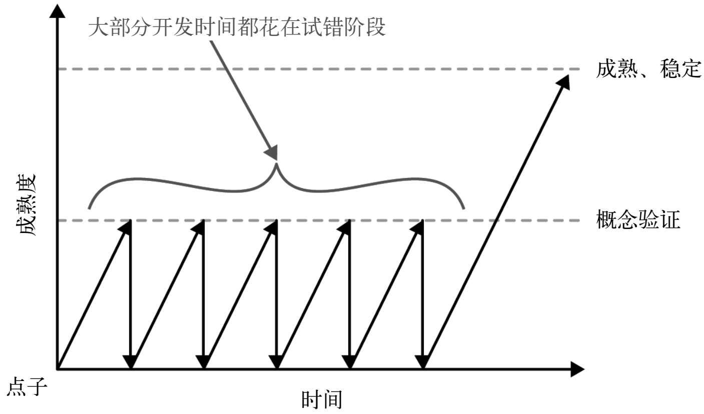


图 2-3：产品开发（试错）

举例说明一下，图 2-4 展示了同一个项目分别采用精益 / 敏捷方法和瀑布方法实现，所制定的不同开发时间线。

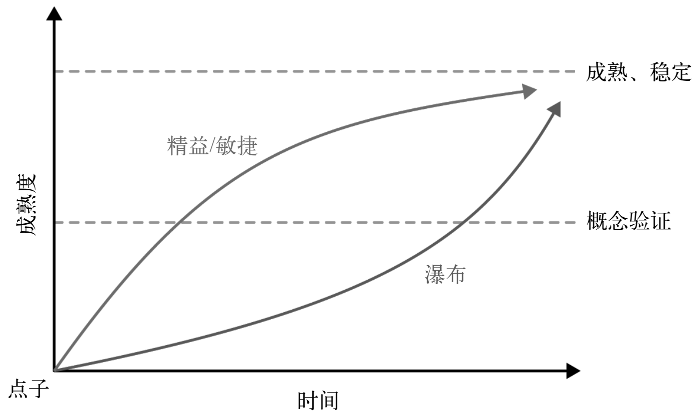


图 2-4：产品开发的敏捷 / 精益方法和瀑布方法的对比（理想情况）

精益和**敏捷**方法的核心原则就是尽可能快地把可用的产品放在用户面前，即便产品离最终完成还有很远的距离。相反，**瀑布**方法则是希望先做出完整的解决方案，再呈现给用户。在理想情况下，从长期来看，用这两种方法去实现功能齐全、成熟的产品所需要的时间大体上是差不多的。但在实际中，多数项目在“概念验证”阶段就止步不前了，如图 2-5 所示。

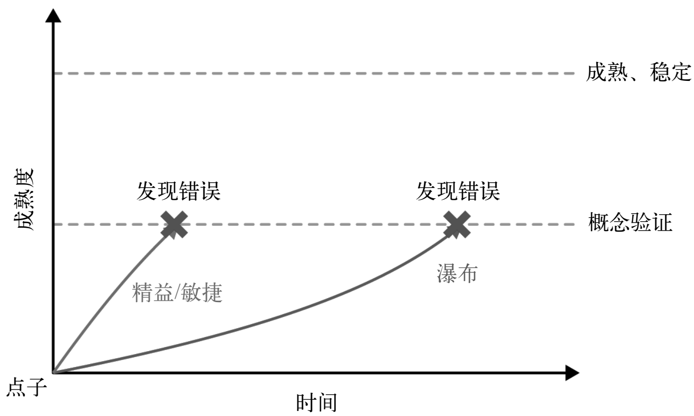


图 2-5：产品开发的敏捷 / 精益方法和瀑布方法的对比（实际情况）

从中可以发现，在这一过程中，用敏捷和精益方法会更早地发现错误；反之，如果使用瀑布方法，就要做出一个完整的产品，才可能意识到它是错误的——这将花费大量的时间和金钱。假如你对一个产品已经做了大量的投入，编写了许多代码，把它抛弃会很困难，也会让你更加泄气。但是，你又不得不把它丢掉，重新回到绘图板前。不仅如此，你可能会多次经历这样的过程，如图 2-6 所示。其中展示了不同方法发现错误（普遍的说法是获得真实用户的反馈）所需时间的差异，这正是速度制胜这句话的意义所在。

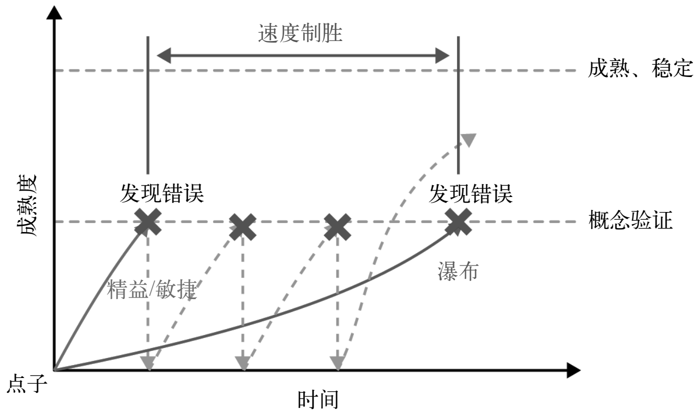


图 2-6：产品开发的敏捷 / 精益方法和瀑布方法的对比（速度制胜）

我喜欢用**速度制胜**这个词，因为它又短又好记，但是用**频率制胜**可能更为精确。这并不是说要像变魔术一样只用一半的时间去完成相同的工作量，而是说要安排好工作，尽快得到反馈。这是因为反馈回路短的系统通常总是胜过回路长的系统。例如，更快的反馈回路可以改善（提高）你的价值曲线：比起要持续五年投钱才能知道行不行的项目，那些一个月后情况就已经清楚的项目更容易吸引别人投钱进去。快的反馈回路也可以改善你的代码：五分钟前才引入并且被自动化测试（阅读 7.2.1 节了解更多信息）立即发现的 bug，和五个月前引入并且是由于顾客抱怨才发现的 bug 相比，前者的修复成本更低，修复可行性也更高。

在软件领域之外，快的反馈回路所带来的好处也同样存在。例如，美国空军奇人约翰 • 博伊德上校在讲解战斗机空战时也说出了相同的道理。如果有两种战斗机，一种性能更出色（爬升更快、转向更好，视野也更好）但无助力，而另一种有液压助力，控制起来更省力，那么后者在实战中反而更有优势。在空战中，双方飞行员通常都是按照所谓的 OOPA（observe, orient, plan, act，即观察、确定方向、制订计划、行动）的顺序进行操作，而液压助力能够让它的飞行员以稍微快一小点的速度完成 OOPA 的过程。

> *博伊德确定赢得空战的主要决定因素并不是更好的 OOPA，而是更快的 OOPA。博伊德提出，迭代的速度会打败迭代的质量。*
>
> ——Roger Session

最后这句话正是**博伊德法则**：迭代的速度会打败迭代的质量。这是一条出人意料的法则，但当我们在处理复杂、不可预知、总是混乱无序的系统时，这条法则就很有意义了。这和投资有点儿像：把钱分散投在多种多样的股票上实现多元化投资组合，与把所有钱都投在一只股票上相比（就像把所有鸡蛋放在一只篮子中），前者的成功率更高。

无论是博伊德法则、速度制胜、敏捷还是精益，它们背后的基本理念都是：我们有一些设想是错误的。问题是，我们不知道哪些是错误的[15]。也许你做的是错误的东西，也许你

注 15：被誉为“现代广告之父”的约翰 • 沃纳梅克曾经说过：“我知道我有一半的广告费都浪费了，问题在于我不知道是哪一半。”

把东西推销给了错误的受众，也许你的商业模型是错误的。要做一个成功的产品，就必须分辨出错在什么地方，然后解决问题。在创业领域，一切都在快速变化，唯有连续不断地重复“发现问题－解决问题”这样一个过程。

> 你知道人们说过，从加利福尼亚飞往夏威夷的飞机 99% 的时间都是偏离航线的，只是在不断修正吗？成功的创业也是如此，除非他们出发时就大错特错，朝着阿拉斯加飞去了。
>
> ——Evan Williams，Blogger、Twitter 和 Medium 创始人

如果你有无尽的时间和预算，就可以按照任意顺序和速度去做产品。但是，现实中的创业只会给你一条长度有限的跑道。所以，尽可能快、尽可能低成本地对风险最大、最根本的假设进行测试，就成了一场比赛。但不要把它曲解为投机取巧、半吊子编码、忽略所有最佳实践、发布平庸产品，或者在临时用胶带、黏合剂拼凑而成的组装电脑上去修改程序。有时候你需要走捷径，但是捷径并不是目标。我们的目标是要找出一些尝试的方法，能够以最低的成本学到最多的东西。

对某些产品而言，即便最小的尝试方法也需要有相当优美的体验；对于其他一些产品，只要有骨架就足够了。某些情况下，你根本不需要实现产品，3.2 节谈论这个问题。一般的规则是要遵循“完成比完美更好”原则。否则，就像 Reid Hoffman 所说的：“如果你第一次发布的时候没有感到尴尬，就是产品推出的时间太晚了。”即便如此，有少数领域并不是依靠速度制胜的。当我们在处理法律、安全、隐私或资金问题的时候，应当花些时间，遵循“测量两次再动手”原则。

到目前为止，我们学到了以下内容：大部分想法都是不可行的，大部分的假设都是错误的，我们经常要和时间赛跑，产品十有八九是失败的。虽然胜出的概率不大，但是有一些方法可以提高这一概率，那就是客户开发。

2.2.2 客户开发

开发产品的传统方法就是把开发人员、产品经理和设计人员封闭在一座大楼中，给他们一年的时间，等到他们完成工作，就尝试把产品卖给客户。问题是，正如前面所说的，你的许多假设其实都是错误的，所以这样一种产品开发过程最终是不可能成功的。假设一般都不是错在如何实现产品或者使用什么技术上，而是搞错了想要这种产品的客户。CB Insights 对超过 100 家创业公司的调查分析研究表明，创业失败的首要原因，而且也是遥遥领先的第一原因，就是“没有市场需求”。

在《四步创业法》一书中，Steve Blank 描述了一种叫**客户开发**的过程，该过程应该和产品开发过程同时进行。我们应该在第一天就把客户纳入开发过程当中，而不是等到产品完成了，才考虑它有没有客户。通过这种方法，可以不断对假设进行测试，在开发的每个阶段都能快速获得反馈。尽管你很想把所有时间都花在产品和技术上（程序员尤其喜欢这样做），但确保你的努力不会白白浪费的唯一方法就是在客户身上花些时间。

> 在创业公司中，只有观点，没有事实。
>
> ——Steve Blank，《四步创业法》

客户验证是指，承认我们的产品点子只不过是需要测试的、未经证明的假设，这样的测

试必须以尽可能快、尽可能低的成本对真实客户实施。越早向同事之外的人验证你的点子，成功的概率就越大。那么，如何找到客户并与之对话呢？可以分解为以下三个连续的阶段[16]。

第一步：验证问题

确保找出客户实际面临并且痛苦到愿意掏腰包去解决的问题。

第二步：验证 MVP

实现潜在解决方案的最简可行产品（minimum viable product，MVP），让少量客户购买该产品进行验证。

第三步：验证产品

把 MVP 完善为完整的产品，让更多客户去购买，对可扩大化的商业模式进行验证。

需要注意的是，上面的每一个步骤可能需要重复多次，甚至要退回到前一个步骤。例如，可能要多次尝试才能发现真正的问题。如果发现了一个问题，可能要多次尝试才能做出客户愿意购买的产品原型。有时候，还会出现找不到可用原型的情况，因此必须寻找新的问题去解决。与之类似，我们可能要进行很多的尝试，才能把原型扩展成为产品。偶尔甚至会发现原型是没问题的，但商业模型却是不可扩展的，这时只能回到前一个步骤。

本章接下来的内容将更深入地探讨如何对问题进行验证。第 3 章会谈谈如何围绕点子去设计一个 MVP。第 4 章会讨论如何利用数据和营销将 MVP 扩展为完整的产品。

2.2.3 验证问题

大量产品的出现都是为了寻找某种问题的解决方案。Segway 和 Google Wave 就是两个有名的例子，但是还有一个大家没怎么听说过的例子—— 一家名为 Patient Communicator 的创业公司。该公司建立了一个在线门户，病人可以在上面和医生联系，医生则可以管理病人的信息。如果你曾经为了看医生而长时间排队，特别当只为了问医生一个小问题时，你就会感受到这似乎真是一个问题。也许可以把问题表述成"医生没有高效的方式去管理与病人的沟通交流和信息"。但是，Patient Communicator 的创始人 Jeff Novich 好不容易才认识到，这并不是一个真正的问题。

> 我们的全部努力——包括数以百计的推销电话、成千上万的 email（因此还被邀请参加"Top 20 客户"晚餐，和 Tout 的创始人 TK 一起共进晚餐[17]）、多场讲座、上福克斯新闻直播、做广告，同时利用自己的人脉以及 Blueprint Health[18] 导师的圈子，最终的结果是得到了一位付费医生用户（后来他也停业了），还有就是与一家有 20 年电子病历（EMR）行业经验的小公司达成了合作协议。
>
> ——Jeff Novich，Patient Communicator 创始人

Novich 列出了公司失败的几个原因，其中之一令我觉得颇受启发："医生想要更多的病人，而不是更有效率的诊所。"其中的差别听上去似乎很微妙，因为高效的诊所就会有更

注 16：这是 Steve Blank 在《四步创业法》中描述的客户开发过程的简化版本。

注 17：Tout 是一家提供 email 发送服务的公司，Tawheed Kader 是该公司创始人。——译者注

注 18：Blueprint Health 是一家总部位于纽约的互联网医疗创业加速器。——译者注

多的病人，但是关注错误的问题会导致整个公司误入歧途。

我们不妨想想牙科行业。多年来，该行业关注的产品和市场营销都是围绕着**抗击牙龈疾病**和**防止蛀牙**开展的。突然有一天，一些市场营销天才意识到顾客真正在意的问题其实是如何让牙齿变白和让口气清新。当然，抑制牙龈疾病和蛀牙也可以获得洁白的牙齿和清新的口气，但是关注了错误的问题就意味着所有的产品、市场营销策略和促销材料全都是错误的。如果你关注的是蛀牙，可能就想不出一些赚钱的产品点子，比如美白牙贴、薄荷糖、漱口水和 3D 美白牙膏（管它是什么东西呢）。这就是我们为什么要在着手解决问题之前先验证一下我们是不是识别出了真正的问题。正如哈佛大学商学院营销学教授 Theodore Levitt 所说的："人们并不想买 1/4 英寸的钻头，而是要钻出 1/4 英寸的孔！"

对于你找出的问题，验证它是否大到有必要建立一家创业公司去解决，这点也是十分重要的。在思考问题的大小时，有三个方面需要考虑：频率、密度和痛苦程度。

- 频率：你所解决的问题经常发生吗？
- 密度：有很多人都会面临这个问题吗？
- 痛苦程度：该问题只是让人讨厌，还是绝对必须解决？

——Manu Kumar，K9 Ventures

我们来看看 Facebook 和 LinkedIn。Facebook 在频率（每天你都会多次用它和朋友或家人交流）、密度（几乎所有上互联网的人都会使用）方面得分非常高，但是在痛苦程度方面就比较低了（与家人、朋友交流还有许多其他的方式，比如当面沟通、电话、短信、即时消息软件、博客、Skype、Twitter、SnapChat，等等）。另一方面，LinkedIn 在频率上得分比较低（你不需要经常更新个人资料、找工作或在上面建立关系），在密度方面得分中等（所有专业人士都可以使用），在痛苦程度方面得分较高（在找工作、找候选人或联系同事方面没有其他更好的方式）。

并不是所有的产品都像 Facebook 和 LinkedIn 这样可以显而易见地做出评价。公平地说，这些社交网络平台的潜在规模在早期并不是很明显，如果要评估问题的规模，就必须进行市场规模的估算。

1. 市场规模估算

市场的规模决定了你可以赚到多少钱，由此可以筹集到多少资金，公司规模可以发展到多大，可以实现何种产品，可以采用什么样的销售和营销策略，以及其他一系列因素。考虑市场规模有一个好方法，就是考虑建立一家赚得 10 亿美元收入的公司的几种方法。

- 以 1 美元的价格销售 10 亿件产品：可口可乐（罐装汽水）；
- 以 10 美元的价格销售 1 亿件产品：强生（家用产品）；
- 以 100 美元的价格销售 1000 万件产品：暴雪（《魔兽争霸》）；
- 以 1000 美元的价格销售 100 万件产品：联想（笔记本电脑）；
- 以 1 万美元的价格销售 10 万件产品：丰田（汽车）；
- 以 10 万美元的价格销售 1 万件产品：Oracle（企业级软件）；
- 以 100 万美元的价格销售 1000 件产品：Countrywide（高端金融抵押公司）。

——Balaji S. Srinivasan，斯坦福创业项目工程课程

如果你认为产品大概值 10 美元，要以此创建一家能赚取 10 亿美元收益的创业公司，就至少需要有 1 亿人的市场（或者还要更大，因为你要和竞争对手分享这个市场）。你需要预先拥有许多本钱（因为销售每份产品所赚的钱相对较少，而且对于某种产品来说，要拥有数百万的用户需要花很长一段时间）；此外，还需要考虑一种能实现如此巨大的用户量的营销策略，比如广告。另一方面，如果你认为产品值 10 万美元，期望的市场就至少是 1 万的客户，也许可以利用早期少量的客户把生意发展起来，你可能需要更多地关注如何建立一个庞大的销售团队，而不是使用广告。

下面，我会列出评估市场规模的几种方法。

广告

许多广告公司都会提供一些广告目标分析工具，我们可以在不需要购买任何广告的情况下对市场进行研究（虽然购买广告是测试 MVP 的好方法，3.2 节将详细讨论）。例如，我们可以用 Google 的 AdWords Keyword Planner 研究每个月有多少人搜索某些特定术语。我在对 hello-startup.net 做研究的时候，查阅了大概 50 组相关关键字（例如“创业点子”“代码评审工具”“净值计算器”），发现平均每月每个关键字都有超过 1200 万次的搜索。这给了我信心，“如何创业”确实是一个真正的问题。而其中的资源页面也可以帮助我雕琢语言，例如我发现人们也经常使用“商业点子”来代替“创业点子”。我还使用过其他几家公司的广告工具，发现 Facebook 上大约有 1600 万人对创业感兴趣，Twitter 上有 200 万人对创业感兴趣，LinkedIn 上则有 1300 万人把他们所在的行业列为创业领域。

竞争

如果已经有公司在解决你发现的问题，其实未必是坏事。甚至可以说，“你的想法并不唯一”才说明你发现了真正的问题。要寻找你都有哪些竞争者，可以使用前面介绍的广告工具，找到合适的关键词，试着在 Google 和一些移动应用商店中搜一搜（应该不难找到，否则他们的客户也就无法找到他们了，如果真的找不到的话你也就不用担心竞争了）。如要想了解某个特定的竞争者正在做什么，你可以试试用网站分析工具（例如 comScore、Quantcast）和移动分析工具（例如 App Annie、Xyo）去估算他们的流量。你也可以使用 CruchBase 或 AngelList 这样的网站，看看竞争者获得了多少投资以及背后是哪些投资者。

我曾经考虑把 hello-startup.net 做成一个移动应用，所以对竞争者做了些研究。通过 Google，我找到了其他几个包含创业资源的应用，比如 Elevatr、Tech Startup Genius 和 Crazy About Startups。通过 Xyo，我发现这些应用都没有多少吸引力，其中做得最好的 Elevatr 大概有 17 000 次安装。我也搜索了其他图书的配套应用，发现其中有几个应用有较大的吸引力，比如 George R.R. Martin 写的 A World of Ice and Fire 的应用（免费，420 000 次安装）以及 Mark Bittman 的 How to Cook Everything 的应用（5 美元，230 000 次安装）。这些信息让我大概了解到这样的应用可以获得多少安装量（少则几千，多则几十万），甚至它们之间不同的定价策略（免费或 5 美元）。

社区

验证问题还有另一种好方法，就是看看社区中是不是已经有人在讨论这些问题了。你可以在聚会、会议、用户组和在线论坛等网站上搜索，估算一下这个问题影响了多少人。例如，在研究 hello-startup.net 的时候，我在 meetup 网站上看到有 15 000 个创

业小组（400 万成员）、3000 个科技创业小组（100 万成员）和 2200 个精益创业小组（650 000 成员）。在 lanyrd 网站上，我发现有 119 个创业会议，并向其中的几个会议提交了申请，得以和这些社区中的人进行实际的交流。我也在 subreddit 上寻找有关创业的内容（大概涉及 74 000 名会员），在 LinkedIn 上寻找创业和创业者小组（大概涉及 150 000 名会员），在 Quora 上查找有关创业的主题（大概涉及 800 000 名关注者）。当然，我也在 Hacker News 上搜索（每天至少有 120 000 名独立用户阅读了有关创业的内容）。

市场研究和报告

某些传统的研究方法也是值得尝试的。我们可以试着在网上搜索探讨你所关注主题的报纸、图书、期刊、课程、广播和博客。如果有必要，你也可以查阅美国证券交易委员会的备案文件或政府报告（例如查阅美国小企业管理局的相关报告）。我在研究 hello-startup.net 的时候，发现了数以百计的博客都在关注创业（例如 Paul Graham 的随笔、TechCrunch 和 OnStartups），还有几十本书（例如《创业者》《精益创业》和《创业者手册》）以及好几门课程（例如斯坦福的“How to Start a Startup”以及 Coursera 的“Startup Engineering”）。

目前也有一些公司专门针对特定的行业收集相关数据并发布报告。其中有些数据是免费的，比如世界银行数据。另外，也可以花钱请 Nielsen Media Research 这样的公司为你进行市场研究，或者找 AYTM 那样的公司代表你向目标客户发送调查问卷。

产品数据

如果产品已经面世，我们可以收集到许多数据并进行分析，对产品新特性的影响进行评估。这方面内容将在第 4 章详细介绍。

上面的这份列表并不全面，但已经足够开始一些估算，从而对市场规模进行评估。只要找出一定规模的问题，下一个验证的步骤就是找出一小部分面临该问题的客户并亲自与之交谈。

2. 与真正的客户交谈

我们和客户交谈，目的是要尽可能地了解他们的日常生活，对下列问题做个决定。

- 对该客户而言，那是一个真正的问题吗？
- 针对该问题，有什么可能的解决方案？
- 该客户愿意支付多少钱去解决这个问题？

要回答这些问题，我们需要走出去和真正的客户交谈。但这样也有一个问题：直接询问客户需要什么，得到的答案一般都不太让人满意。有些客户根本就不知道他们自己想要什么。有些客户虽然知道自己想要什么，但是他们会心口不一，告诉你要 X，实际上想要的是 Y。有时候这是因为他不想伤害你的感情，所以会说喜欢你的产品，尽管他们知道自己永远不会买这样的东西。有时候，客户的主观意愿就是不想让你知道真相，比如不想告诉你他们愿意为某个产品支付多少钱。有时候，只是因为客户并不知道自己要选择什么。

> 如果我问你们，比如说，就在这房间里的人，你们想要什么样的咖啡，你知道自己会怎么回答吗？你们每个人可能都会说：“我要香浓碳烧黑咖啡。”这就是你问人们想要什么咖啡时经常可以听到的答案。你喜欢哪种呢？香浓碳烧黑

咖啡！实际上，真正喜欢香浓碳烧黑咖啡的人占百分之多少呢？根据 Howard Moskowitz 的研究，这样的人大概只占 25%~27%，多数人喜欢的是奶味淡咖啡。但当别人问你想要什么口味的时候，你从来、也永远不会对别人说：“我要奶味淡咖啡。”

——Malcom Gladwell，TED 演讲 Choice, Happiness and Spagetti Sauce

即便客户完全知道自己的需求是什么，即便他们愿意诚实地面对你，大多数时候，你也仍然无法找到好的解决方案，因为客户通常只会考虑更好、更快、更便宜的那 10%。许多客户会向你提供一些明确的功能需求，但我们的目的并不是要带走这样一份长长的功能清单。一小点增加的改进和稍好一点的功能很难成为好的创业点子，这一点我们将在 3.2.2 节讨论，我们的真正目标应该是获得对潜在问题的深入理解。

如果我问人们想要什么，他们肯定会说想要跑得更快的马。

——亨利·福特

要得到潜在的问题，我们必须更多地去倾听和观察，而非交谈。不要把自己的想法强加在客户身上，或者尝试去说服他们。相反，我们要让他们尽可能地多谈些东西。你可以用一种经典的技巧，这是由丰田的创始人丰田喜一郎所提倡的，就是**五个为什么**，这个技巧最好是用一个例子来说明。假设你是一家运输公司的老板，一位叫 Bob 的员工告诉你，他的卡车无法启动了。这时，你要做的不是直接去寻找解决方案，而是要反复地询问“为什么”，从而找到根本的原因。

Bob：卡车无法启动了。

你：为什么？

Bob：电池没电了。

你：为什么？

Bob：（调查）好像是发电机不工作了。

你：为什么？

Bob：（调查）发电机的皮带坏了。

你：为什么？

Bob：（调查）发电机皮带真是太旧了，很久以前就该换了。

你：为什么？

Bob：我猜是我们没有完全按照维护计划去做保养。

如果你解决的是听到的第一个问题，即卡车无法启动，你的解决方案可能就是换掉卡车或者电池——但这只是处理了一个症状。问了这五个为什么，你就可以揭示潜在的问题其实是车队中的卡车没有完全按照维护计划进行保养。当然，我们并非总是要精确地问五个为什么才能找到根本的原因，但至少应该去问上一次，确保自己找到真正的问题。

和一些客户进行交谈之后，你对什么是真正的问题应该会有更好的理解。在你一头扎进去解决问题之前，最后还要核实一个问题：可行性。

3. 可行性

是否能够解决一个问题存在两个因素。

- 问题可以被解决。
- 问题可以被你解决。

第一个问题和市场实际情况有关。我们既要考虑前面所提到的市场规模、问题验证，也要核实解决该问题的技术是否已经存在，还要看解决方案从经济上是否足以建立起可盈利的商业模式。红杉资本是当今世上最成功的风险投资公司之一，它的合伙人会询问创始人们一个问题："为什么是现在？"世界发生了什么变化，使得现在成为建立这家公司的最佳时间？你知道什么其他人所不知道的？为什么没有人在两年前建立这样的公司？为什么两年以后再建立这样的公司就太迟了？

例如，Webvan 是一个在线杂货店，它在 2001 年的破产使其成为最著名的互联网公司失败案例之一。在此之前，该公司已经烧了 8 亿多美元去建立仓库和自己的运输车队。现如今，又冒出了许多新的日用品派送创业公司，比如 Instacart 和 Postmates，看起来似乎比 Webvan 做得更出色。如果问他们"为什么是现在"，其中有一个因素就是现在的顾客远比 15 年前更习惯在网上购买东西，而且近年来出现的智能手机、无线数据、GPS 连接技术使这些公司可以利用司机的个人车辆去组建运输车队，或者利用现有的杂货供应商去建立库存。

如果该问题可以被解决，对可行性的下一个验证就是你是否是解决问题的合适人选。这个问题在一定程度上和你的个人资产有关，这些资产包括但不限于经济实力（例如现金和财产），也包括你的技能、知识和人脉，这也是解释领域专业知识为什么如此重要的另一个原因。另外还有一个重要的因素，就是这个点子是不是你真正所在意的。我在第 1 章提过，建立一家成功的创业公司需要花费十年左右的时间，需要付出巨大的辛勤工作和牺牲。所以，我们不单单要找到可以解决的问题，这个问题还必须是自己乐意花上生命中的下一个十年去解决的。

2.3 小结

2000 年 5 月 24 日，Timothy Gowers 在克雷数学研究所的千年会议（Millennium Meeting）上做了一个名为"The Importance of Mathematics"（数学的重要性）的演讲。Gowers 形容著名的剑桥数学家哈代"对他所选择的领域——数论，在当前和可预见的未来都没有什么应用深感满意，而且非常自豪。对他而言，数学的价值主要就在于体现它的美"。许多数学家更喜欢因为问题固有的美而去研究问题，而非这些问题有什么实际的好处。尽管如此，数学已经成为不计其数、有着巨大实际价值的各种发现的基础，物理学家、化学家、工程师、程序员和其他无数的人，每天都在使用数学去实现现代社会的各种工具和技术。通常来说，数学越是美丽，它在现实世界中就会越有用。即便是看起来似乎纯粹就是为了数学而数学的数论，也可以有许多实际的应用，比如 RSA 加密，这是我们可以在互联网上安全地交换密码和信用卡信息的原因所在。对此，哈代应该要失望了。

所有的数学概念都有着紧密的联系，这些联系通常是以不可预知的方式发生的。Gowers 宣称，我们无法知道数学的哪些部分在现实中是有用的，哪些部分又是无用的，如图 2-7 所示。因此，我们最好鼓励人们去研究所有的数学，即便（或者特别是）研究它的出发点是对数学之美的追求，而非其实际价值。

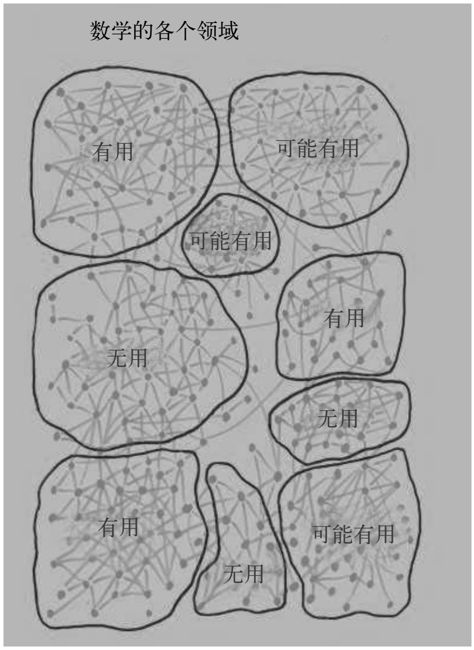


图 2-7：Gower 的数学知识图示，“有用”和“无用”的知识是无法区分开的

对于点子来说也是类似的：没有能预测哪些知识可以让你产生有用的点子，哪些知识不能的方法。最好的做法就是尽可能多学习一些东西，特别是你觉得有意思的主题。换句话说，“获得创业点子的方法就是不要去想创业点子”，而是把自己变成一个有创业点子的人。找到吸引你的主题，花大量的时间去思索，在点子日记上写下自己的想法并与他人分享。学会利用约束条件，寻找痛点，多出去走走，为你的潜意识提供大量时间处理所学到的东西。最终，便会萌发点子。

在此阶段，点子仍然是不成熟的。所以要注意，不要因为太快对这个点子下结论而扼杀了它，就像你无法预料哪些数学上的概念是重要的，你也同样无法预测哪些点子在未来会有更大的发展。回到 1997 年，Larry Page 也不知道 Google 会是多么大的一个点子，那时他还想把公司以 160 万美元卖给 Excite（今天 Google 的价值大约是 4000 亿美元）。我在这本书中采访的每一个程序员，都没有想过他们的创业公司会发展得那么大。Jessica Livingston 在《创业者》一书中采访过的所有创始人都是如此，包括 Max Levchin（Paypal）、Caterina Fake（Flickr）、Craig Newmark（Craigstlist）和 Steve Wozniak（Apple）。

> 最出乎我意料的是，这些创始人非常不确定他们实际能把事情做到多大，有些公司甚至就是因为一个偶然的机会才建立的。大家都认为创业公司的创始人拥有着某种超乎常人的信心，其实最初很多创始人对于公司的创立是不确定的。他们所能确定的只是可以把事情做好——或者尝试去解决某些存在的问题。
>
> ——Jessica Livingston，《创业者》

当你的某个点子最终发展成某种激动人心的事物时，请在你一头扎进实现之前先进行验证。你可以使用市场研究工具去评估市场的规模，如果结果看起来有希望，就走出去和

真正的客户交谈，至少找十个会买你产品的人交谈。虽然这样看起来似乎很愚蠢，特别是你的产品点子涉及的是数以百万的客户时。但如果你都找不到前十个表示会购买的用户，你的点子就只不过是哄哄人而已。如果你找到了你的前十个客户，并且验证了你的想法是可行的（为什么是现在，为什么是你），这就表明现在是时候把它变成一种产品了，这也就是第 3 章的主题。

创造力并不是一种天赋，它只是一种行事方式。

——John Cleese

第 3 章

产品设计

上一章介绍了如何想出创业的点子。本章将讨论如何围绕这些点子去设计产品。本章前半部分会解释为什么设计对于任何职业都是一项必备技能，并介绍一些大家都得知道的工具和技术；后半部分会关注创业公司的产品设计过程，实际上就是介绍最简可行产品（minimum viable product，MVP）的实现。

3.1 设计

> *在顾客看来，界面就是产品。*
>
> ——Jeff Raskin，《人本界面》

对于上网的人来说，Google 就是一个文本框和一个结果页面，他们不会想到抓取网页的机器人、页面评级的算法，以及分布在世界各地多个数据中心的数十万台服务器；对于需要打车的人来说，Uber 就是手机上的一个按钮，按下去就可以订车，它不是什么实时调度系统，也不是支付处理系统，他们看不到 Uber 为招募司机、与监管者斗争所付出的全部努力；对于使用智能手机的人来说，iPhone 就是由他们所能看到的（例如屏幕）、听到的（例如来电者的声音）和碰到的（例如按钮）的各个部件组成的，它没有 GSM、WiFi 和 GPS 无线收发装置，也没有多核 CPU、操作系统和提供这些部件的供应链，更没有组装这些部件的工厂。对于顾客而言，产品的设计就是他们所在意的一切。

Joel Spolsky 把这种情况叫作**冰山的秘密**。我们所看到的冰山在水面上的那部分只占它总体积的 10%；同样，我们可以看到和触碰到的产品的那部分——用户界面，只占全部工作的 10%。所以，这个秘密就是**大多数人并不清楚这一点**。当人们看到很糟糕的用户界面时，他们会认为该产品的一切都很糟糕。如果你要向潜在客户进行演示，你就得精心雕琢你的展示结果，这才是最重要的。你不能让客户去想象产品将来是怎么样的，现在只要关注“功能”就行了。如果界面的像素看起来很糟糕，人们会认为产品可能也是很糟糕的。

你可能会认为，冰山的秘密不会发生在程序员身上，但其实无人能够幸免。与 Android 相比，我更喜欢 iPhone；与那些只有纯文本 Readme 文件的项目相比，我更喜欢有着精美文档页面的开源项目；与 Blogger 上的博客文章相比，我也更喜欢发布在 Medium 上的博客文章。我们似乎天生就会通过一本书的封面去判断它的好坏。但是，产品设计不仅仅是封面，还涉及印刷、书名、封底的赞誉、排版、布局，甚至是文本本身。

> 大多数人错误地认为设计就是东西看上去的那个样子。人们认为设计就是外观，比如设计师接到一个盒子，被告知**把它弄好看**。这不是我们所认为的设计。设计不仅仅是看上去的样子，还关乎它如何使用。
>
> ——史蒂夫·乔布斯

设计关乎产品如何使用。没错，iPhone 比其他大多数智能手机都更加漂亮，这为它增色不少，但 iPhone 的出色之处不仅在于它的款式。它清晰的屏幕、排版和布局使文本变得更容易阅读；它的按钮很大，易于使用；它的触摸屏非常精确，UI 又快、反应又灵敏。iPhone 可以预测你的需求，根据周围环境的光线自动调整屏幕亮度，或者在你拿着手机靠近耳边时将屏幕完全关闭。你在使用中完全不需要考虑它，不需要跟它较劲，它就是好用。虽然其他智能手机也许在功能或价格上不比 iPhone 差，但它们仍然无法达到这种体验水准。这就是 Apple 如此看重设计的原因，自然而然，这也将 Apple 造就成了世界上最具价值的公司。

即便你不做产品，即便你的职位中并没有“设计师”这个词，设计对你也颇为有用。每个人时时刻刻都会用到设计，无论你是制作幻灯片进行展示介绍、编写个人简历、制作个人主页，还是安排客厅家具的摆放位置、准备上课的教学大纲或者设计软件系统的架构，都要用到设计方面的知识。从根本上说，设计就是如何去呈现信息，让他人可以理解并使用这些信息。人生中的许多次成功其实都取决于我们能够在多大程度上良好地交流，如果在大多数人的教育中加入一点设计方面的训练，结果将大为不同。

正是因为我缺少这样的训练，我过去总认为设计和美术方面的能力只能是天生的。我自己在美术方面的才能仅限于能够在笔记本上画几个火柴棍小人——很明显，我并不具备这样的才能。我花了很长时间才意识到，设计和美术其实都是一种可以通过迭代而学会的技能。

3.1.1　设计是迭代的

几年前，我和妹妹参加了一门美术课程。美术老师是我们家的一位朋友，他过来我们家，让我们用铅笔和水彩画一些静态的实物和建筑物轮廓。有一天，我正在画一幅水果的静态写生，努力为一只橙子涂色。可我尽了全力，也只是把一些单调的橙色涂到一个模糊的圆形中。老师注意到我有点受挫，于是问我：“橙子是什么颜色的？”我不敢确定这个问题是不是有什么陷阱，就试着回答：“橙色？”美术老师笑着说：“还有呢？”我盯着水果看了一小会儿，回答：“我觉得在表面照射到光线的地方，还有一点白和黄。”美术老师的笑容仍然挂在脸上，对我说：“很好，还有其他颜色吗？”这回我花了更长时间去观察水果，“没有别的颜色了，就是只有这该死的橙色，我看到的只有橙色和不同深浅的橙色。”

美术老师探过身来，拿走我手中的画笔，开始修改我的画作。他一边画，一边向我解释着他在做什么：“球体会有一些地方是更亮的，我们可以用白色和黄色来上色。此外还有

阴影，可以用红色、棕色和绿色。橙子也会在一侧投下影子，我们可以用灰色和蓝色来表现，然后再用上一些棕色和红褐色，把橙子的边缘和它的阴影区分开。”（见图 3-1。）

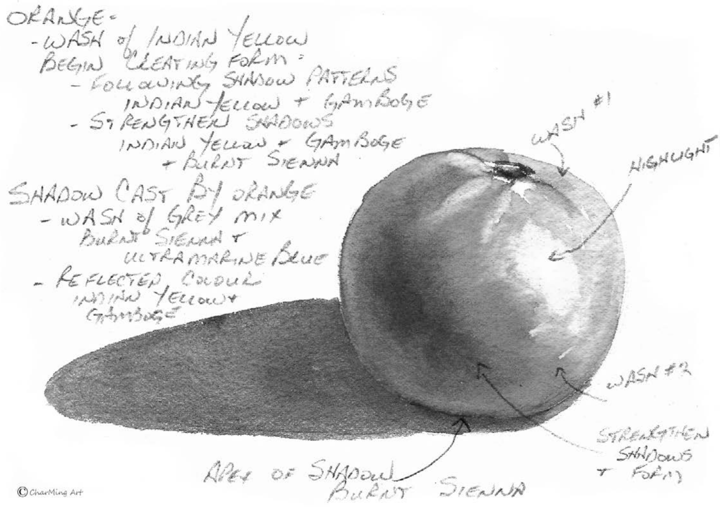


图 3-1：如何画一个橙子。图片由 Charlene McGill 提供

我在画布和实际的水果之间来来回回地观察。橙子并不是橙色的，它是橙色、黄色、白色、红色、棕色、红褐色、绿色和蓝色的。过了一会儿，我有了以下几点认识。

- 美术包含了许多可以通过学习而掌握的具体手段和技术。美术老师了解画出球状物体的所有要素，几乎就像拿着一份菜谱做菜：拿几杯底色，混入一汤匙阴影，加入少许高光，搅拌，球体就做成了。
- 在我脑海中的橙子和现实当中的橙子是不一样的，但我并没有觉察到丢失的各种细节，直到我尝试在画布上复制出橙子的图像时才发现了问题。
- 与之类似，橙子在画布上的样子和现实世界中的橙子也是不一样的。而这样的差异通常都是有意为之的，因为绘画的目的并不是要创作现实中某些东西的照片，而是要用一种特殊的方式把它呈现出来，让人们想到或感受到某种东西。[1]

我离成为画家还有很远的路要走。但理解了画家的思维模式后，我认识到他们的天赋其实也是一种可以提高的技能，我们要训练自己的眼睛去专门观察某些东西为什么看起来是这样的，认识到绘画的目的是要把一些东西传递给观赏画作的人。其中有三个原则可以应用到设计中。

- 设计是一种可以学会的技能。
- 我们必须训练自己的眼睛有意识地识别出为什么有些设计能发挥作用，有些则不行。
- 设计的目的是把一些东西传递给用户。

注 1：有个关于毕加索的经典故事。有一回，他坐火车旅行，一名乘客认出了他，问道：“你的作品为什么是失真的，为什么不把人物画成他实际的样子呢？”毕加索问道：“你觉得他们实际应该是什么样的呢？”这名乘客从他的钱包里拿出了妻子的照片，说道：“看，这就是我妻子实际的样子。”毕加索看了看照片，说：“她又小又扁吗？”

我希望本章能说服读者们相信这三点。对于第一点，我们要记住的最重要的事情就是，设计是一种迭代的过程。第一份草稿可能会很糟糕，但成就好的设计作品、成为出色的设计师的唯一途径就是不断地迭代。设计之所以对我们特别有挑战性，是因为在我们的生活中，无数由专业设计师创作的产品已经把你的品位提高了，我们在早期设计出来的作品是不可与之比肩的。就像一名长久以来都在欣赏莫扎特和巴赫那美妙的小提琴协奏曲的音乐爱好者，梦想着自己有一天能在卡内基音乐厅演奏。当他终于第一次拿起了小提琴，激动地在弦上拉动琴弓时，发出的刺耳声音可能会让自己大吃一惊。每一个从事创作工作的人都会经历一段自己的作品无法满足自己品味要求的时期，这是完全正常的，唯一的解决方法就是创作更多的作品，比如不断地拉小提琴，不断地创作新的设计，不断地迭代。最终，也许要经过很长一段时间，你的技能终于能和你的品位相匹配，这时你就可以创作出自己满意的作品了。但是就目前而言，只要记住完成比完美更好就行了。

第二点，如果想在设计方面更进一步，我们需要有意识地去弄明白为什么某个特定的设计对你是有用或没用的。下次，当你惊讶于 iPad 的使用为何如此简单时，请停下来问问自己“为什么”。它在设计上有什么特点使其简单到几乎所有人都会使用，不管是完全没有科技领悟能力的爷爷奶奶，还是两岁大的孩子。为什么同样的人却无法掌握台式机或者需要触屏笔的平板电脑的使用方法？ 3.1.3 节将讨论我们的眼睛应该关注设计作品中的什么部分。

第三点，即设计的目标是为了与用户沟通，这意味着虽然“看起来漂亮”是设计的一个很有价值的因素，但是更为重要的是要认识到设计是为了帮助人们实现他们的目标。因此，每一次设计都要从理解用户开始，这就是接下来要介绍的主题——**以用户为中心的设计**。

3.1.2 以用户为中心的设计

我记得我和几名同事曾经坐在 LinkedIn 的一间会议室里，准备开始下一个项目。我们知道自己要实现什么，并且把这项工作分解成了一张任务清单。唯一还要做的就是要找个地方把这些任务记下来，以便可以随时跟踪进度。我们决定试试公司其他人都在用的问题跟踪软件，这个软件有各种花哨的功能，比如搜索、报告工具和彩色图表。唯一的一个问题是：我们不知道怎么用。

房间里有七名专业程序员。我们知道自己要做什么，认为自己也知道要怎么做，因为以前我们都已经多次使用过问题跟踪软件，而且一直都在使用网站——不仅如此，我们还会制作网站，而且会做几十年了。所以，我很难表达出当房间里的每个人都被这样一个问题跟踪网站彻底难住时，大家是有多么沮丧。我们花了好几个小时，尝试找出如何在这个程序中定义新项目，定义之后如何开始项目，如何在项目之间移动任务单，如何使用 15 种不同的视图模式，也想弄清楚为什么我们完成项目之后，所有的图表还是空的，还有问题创建界面上的 50 个文本框都是干什么用的。这是让人抓狂的事，经历了各种挫折，我们放弃了，最终还是选择使用便利贴。无论从哪方面来看，问题跟踪软件都比便利贴要好，只有最要紧的一个方面除外：帮助**人**实现**目标**。

注意这里黑体字强调的“人”和“目标”。设计不是按钮、彩色图表或功能特性，它关乎人和目标。在上面的故事中，人就是软件专家，不幸的是，问题跟踪软件无法帮助我们实现跟踪项目工作情况的目标。更糟糕的是，它的失败出现在设计最为重要的目标上。

所有计算机用户的第一目标就是不要让自己觉得自己愚蠢。

——Alan Cooper，*The Inmates Are Running The Asylum*

过去，我设计软件的过程（如果真的可以称之为过程的话）要经过以下步骤。

(1) 和团队成员坐在一起，问："我们的 5.0 版产品要加什么功能才酷？"
(2) 在提出一长串功能之后，辩论一下它们的优先级，很武断地设立一个截止期限。
(3) 在截止期限到来之前，拼命工作完成尽可能多的功能。时间肯定是不够用的，然后开始砍掉一些比较花时间的功能。
(4) 在用户界面上找各种地方，把按期完成的所有功能塞进去。
(5) 向用户发布 5.0 版，希望和祈祷用户喜欢它。
(6) 重复这一过程。

这种过程有许多地方都是有问题的，但最大的问题可能是其中没有一个步骤会考虑用户的真正目标。我做的是"酷"的东西，而不是用户真正需要的东西，我根本没去考虑如何实现用户实际需要的东西（4.1 节会谈到这个问题）。我知道的只是怎么添加新功能，这正是我过去所做的。我犯了严重的"功能病"，花了很长时间才找到治疗方法。

解决的办法就是要认识到我们不能在工程或产品完工之后，才把"设计"加上去。设计就是产品，从产品开发的第一天起，它就应该是其中的一部分。以用户为中心的设计应该纳入我们的产品开发过程中，下面是它的五个基本原则：

- 用户故事；
- 人物角色；
- 情感设计；
- 简单；
- 可用性测试。

1. 用户故事

提前考虑设计并不意味着需要做出一份 300 页的详细规格说明书，而是在一心投入代码开发之前，先定义出用户故事。所谓用户故事，就是从用户的角度简短地描述你所做的东西。它应该回答下面三个问题。

- 用户是谁？
- 他们要实现什么？
- 他们为什么需要？

第一个问题"用户是谁"要求你要理解人，这可是出奇困难的事。你可能觉得以己推人就可以理解人们的行事方式，至少你知道自己的动机是什么。但是看过上一章内容之后，你就知道我们的行为很大程度上是受潜意识控制的，而我们通常是完全没有意识到的（阅读 2.1.2 节了解更多信息）。

如果你是一名程序员，想理解你的用户会更加困难。每个人都会对"一种产品如何工作"形成自己的**概念模型**。但程序员的模型通常是非常细节化的，一般都处于界面、事件、消息、API、网络协议和数据存储这样的层次，而典型的用户模型通常没有那么多的细节，既不精确也不完整（例如，许多用户是不能区分软件与硬件、显示器与电脑有什么差别的）。这种概念模型上的不匹配会导致程序员很难和用户沟通。

其中要把握的关键点是：沟通是设计目的之所在。我们努力把信息呈现给用户，告诉他们可以做什么，向他们展示如何去做。不幸的是，许多程序员并没有意识到，因为对自己的软件非常了解，所以考虑软件的方式和用户是完全不同的，我们全然记不起初学者面对我们的软件是什么感觉。这种情况称为**知识之祸**，是一种认知效应，斯坦福大学的研究给出了漂亮的论证。

> 1990 年，伊丽莎白 • 牛顿依靠一项研究取得了斯坦福大学的心理学博士学位。她研究的是一个简单的游戏，将参与者分为两种角色："敲打者"和"倾听者"。敲打者会得到一份歌单，里面是 25 首著名歌曲，比如《祝你生日快乐》和《星条旗永不落》等。每一位敲打者都挑选出一首歌，把它的节奏敲打给倾听者听（通过敲桌子的方式）。倾听者的任务就是根据敲出来的节奏把歌曲猜出来（顺便提一下，如果你旁边有人可以充当"倾听者"，这个实验在家玩也很好玩）。
>
> 游戏中倾听者的任务相当困难。在牛顿的整个试验过程中，一共有 120 首歌被敲出来，但倾听者只猜出了其中的 2.5%，即 120 首歌中仅猜出了 3 首。
>
> 而这正是这篇心理学博士论文这么有价值的原因所在。在倾听者猜测歌名之前，牛顿询问了敲打者预计倾听者正确猜出歌名的概率是多少。他们预测的概率是 50%。实际上，敲打者每敲 40 次才有一次可以把信息传递出去，但他们认为自己每两次就可以传递出一次信息。为什么会这样呢？
>
> 当敲打者敲出节奏的时候，他的大脑其实正听着那首歌曲。你可以自己先试一下，敲出《星条旗永不落》的时候，你的大脑不可避免地会听到那个调子。与此同时，倾听者却无法听到那个调子——他们所能听到的只是一连串不连续的敲打，就像某种怪异的莫尔斯码。
>
> ——Chip Heath、Dan Heath，《让创意更有黏性》

作为程序员，当你在设计软件的时候，你的大脑其实一直都在"听着歌曲"。然而，你的用户却什么都没有听到，他们必须通过你所设计的用户界面（user interface，UI）去使用软件。你不能期待用户知道你所知道的，你也不能指望用户通过文档或教程来填补这一鸿沟。（正如 Steve Krug 所说的："关于说明书你必须知道的最主要的一件事就是，没有人想读说明书。"）所以，想要做出成功产品的唯一选择就是做出出色的设计。

这听起来也许是再显然不过的事，但程序员却很容易把它置之脑后，而他们工作中所用的工具也可以说是糟糕设计的顶峰之作。某种程度上，这是因为大多数设计给程序员使用的软件也是设计给电脑用的，而电脑可不在乎可用性的问题。从早到晚，我们要设法记住各种魔法咒语（有个老笑话说，"我用 Vim 两年了，主要因为我不知道怎么退出"），学习解析日志文件、核心转储、XML 这样深奥的格式（想要精通 Java，还必须对一门叫堆栈跟踪的语言非常熟悉），还要被错误消息当成卑微的罪犯一样对待（"表达式开头非法""无效的语法""错误代码 33733321""中断、重试、退出"）。要成为一名成功的程序员，就得对糟糕的设计有很强的容忍力，几乎要到熟视无睹的地步。但如果要开发普通人可以使用的软件，就必须和他们一样去感同身受，压制自己作为程序员的许多本能感受。

编程的过程和制作易用产品的过程是格格不入的，简单来说就是程序员的目标和用户的目标是有显著差别的。程序员希望构筑产品的过程顺利简单，用户则希望与程序的交互顺利简单。而这两个目标通常都不会共生于相同的程序。

——Alan Cooper，*The Inmates Are Running The Asylum*

即便你克服了理解用户的障碍，还会面临第二个问题：他们希望实现什么。这个问题依然会成为不少人的绊脚石。最常见的设计错误就是把用户的目标（他们要实现的**是什么**）和任务（他们可以**如何**实现）混淆了。经典的例子来自于冷战时期的太空竞赛，NASA 的科学家意识到无法在太空的微重力环境下使用钢笔，所以花了数百万美元研发出一种带有加压墨盒的钢笔，它可以在零重力、上下颠倒、水下以及高温严寒等各种环境下书写。与此同时，苏联人使用的却是铅笔。这个故事虽然只是个传闻，但它却精彩地说明了当我们无视根本目标，而对做事情的某个特定方法过于关注时，会有什么荒唐的事情发生。正如亚伯拉罕·马斯洛所说："如果你唯一的工具是一把锤子，那么你看到的任何东西都像钉子，我想这对人们是很有诱惑力的。"

把任务从目标中分离出来的一种最佳方法就是使用上一章介绍过的"五个为什么"技巧（阅读 2.2.3 节了解更多信息）。另外还有一种方法，就是遵循 Alan Cooper 在 *The Inmates Are Running The Asylum* 一书中提出的建议。

要区分任务和目标之间的差异，有一种很简单的方法。任务会随着技术的变化而变化，但目标有一种讨人喜欢的属性——它会非常稳定。例如要从圣路易斯到旧金山旅行，我的目标始终是速度、舒适性和安全。但在 1850 年去加州金矿的话，我会在全新的高科技科内斯托加马车中度过这段旅程。为了安全，我还会带上温切斯特来复枪。而在 1999 年从圣路易斯到硅谷，我会乘坐最新的高科技波音 777 飞机。

——Alan Cooper，*The Inmates Are Running The Asylum*

第三个问题"为什么人们需要它"实际上就是强迫你证明为什么你要做你所做的东西，这就是上一章介绍的客户开发过程可以发挥作用的地方（阅读 2.2.2 节了解更多信息）。如果该产品或功能并不能为真正的用户解决重要的问题，你就不应该浪费时间去实现它。

所以，无论什么时候，我们都应该花些时间用书面方式回答这三个用户故事问题，把你的产品点子从脑海中短暂、模糊的想法转变成纸面上具体的文字和图示，这样可以暴露出你在问题理解上存在的一些缺陷。而且，把问题仅仅呈现在纸面上涂涂画画的过程中，比修改已经编写了成千上万行代码的成本低很多。在 Readme 文件、维基系统或便利贴上记录几行文本、画出几幅草图，就可以促使自己从用户的角度去感受这种端到端的体验，确保自己知道自己在做什么，为谁做，以及为什么值得做。

2. 人物角色

这里有另一种可以显著提升设计技能的快捷方法：不要再为"平均的人"设计产品。人平均下来就是不洋不土、不男不女，如果你为平均的每个人做设计，那么谁都不会喜欢你设计出的东西。

真正的平均用户被保存在日内瓦国际标准局的密不透气的地下室中。

——Steve Krug，《点石成金》

设计**人物角色**是一种更好的思路。人物角色就是一个虚构的角色，他代表使用你的产品的一个有特定目标、性格和要求的真正用户。例如，我为 hello-startup 设计了以下几个角色。

(1) 麦克：他是马萨诸塞大学阿默斯特分校计算机科学系的一名 19 岁的大学生。麦克大多数时间都醉心于技术，在中学时就开始编程，每天花很多时间浏览 Reddit 和 Hacker News。他正在考虑毕业以后的工作问题，他对创业感兴趣，但是父母却更希望他加入知名的正规公司，他不知道何去何从。
(2) 莫妮卡：她是 Oracle 公司的一名 28 岁的高级软件工程师。莫妮卡获得了麻省理工学院的计算机科学学位，毕业后又在数家大型软件公司工作，经过多年工作之后最终选择了 Oracle。她现在对工作感到厌烦，正在寻找一些更有挑战性、能够让她对世界有更大影响的事情。她有几个创业点子，但不是很确定接下来要做什么。
(3) 马赫什：他是一名从斯坦福退学的 21 岁的程序员，他和室友一起创立了一家公司。马赫什与合伙人已经在这家公司工作了 6 个月，但还在苦苦挣扎中。他们不是很清楚如何设计产品，应该用什么样的技术，如何让客户去使用他们的产品，或者到哪里找开发人员来帮助他们。

每个人都应该有名字、年龄、简历、工作经历和相关技能、信仰和目标，以及其他一些与你的业务相关的细节。为了让虚拟角色看起来更像真人，可以为每个角色添加一张照片（最好是在图片网站上找来的照片，而不是生活中某个熟人的照片）。为产品定义好人物角色之后，无论在用户故事中，还是在谈话中，都不用再去关注“平均用户”了。团队不用再去争论“平均用户”是更喜欢 X 功能还是 Y 功能，因为每个人对于什么是“平均”都会有不同的理解。相反，我们只需要讨论我们的人物角色是喜欢 X 还是 Y 就可以了。例如，hello-startup 的“平均用户”想要一个计算程序来帮助自己对股票期权进行估值吗？这我也不知道。但麦克、莫妮卡或马赫什需要这样一个计算程序吗？我可以有根据地推测麦克和马赫什会觉得这样的工具是有用的。

应该根据市场研究和客户访谈对人物角色进行设定（阅读 2.2 节了解更多信息）。我们的目标就是要找出这样一小群主要人物（通常是 1~3 个），让产品必须完全符合他们的目标，否则整个产品就是失败的。例如，麦克、莫妮卡和马赫什就是 hello-startup.net 的主要人物，如果他们无法在网站上找到他们所需要的东西，那么这个产品也许就不存在了。而我们的目标就是想办法找出这些主要人物的目标，做出特别能够帮助他们实现那些目标的产品，尽可能地取悦他们，除此以外别无他法（阅读 3.2.2 节了解更多信息）。

> 你关注的目标越广泛，错失靶心的必然性就越大。想让大量人口中 50% 的人满意你的产品，从而实现 50% 产品满意度的目标，这种做法是行不通的。我们只能挑选出 50% 的人，想方设法让他们 100% 满意，才能实现我们的目标。我们甚至可以瞄准市场中 10% 的人，让他们 100% 地**心醉神迷**，从而取得更大的成功。这听起来可能有点违背我们的直观感觉，但为单个用户进行设计是满足广大人群需求最有效的方式。
>
> ——Alan Cooper，*The Inmates Are Running The Asylum*

人物角色之所以是如此强有力的设计手段，是因为它可以促使你去考虑真正的人，估计他们的真实需求、局限性和个性，最为重要的是，考虑他们的情感。

3. 情感设计

研究表明，人与计算机及软件之间的交互在很大程度上与人类之间的交互类似。大多数人会对计算机彬彬有礼，尽管偶尔也会怀有敌意；对于具备女性声音的电脑和男性声音的电脑，他们的反应是不同的，在适当的场景下，人们会把计算机当作团队的成员，甚至是朋友。当打印机无法工作时，你是不是曾经也对它乱发过一通脾气？是不是有过很喜欢一款软件的感觉？有没有在电脑崩溃之后，祈求它别把你的 Word 文档弄丢了？不管你有没有意识到，每款软件其实都会让你有不同的感觉。我们大部分的情感反应是自动产生的，控制这些反应的大脑区域尚未进化到能够对人和举止像人的无生命物体进行区分的程度。

这就是为什么最出色的设计总会有一些人性和情感的因素在其中。例如 Google 有许多隐藏的复活节彩蛋（可以试试用 Google 搜索"recursion""askew"或者"Google in 1998"）、愚人节笑话（比如搜索 PigeonRank 和 Gmail Paper），还有一个"手气不错"按钮。在很多日子里，他们还会替换"Google 涂鸦"的 logo，以纪念一些重要的事件。美国维珍航空在他们的航线上把标准、无聊的飞行安全视频替换成有趣的音乐视频，该视频目前在 YouTube 上已被观看过 1000 万次以上。放假的时候，Amazon 会在网站上增加一个音乐播放器，让人们在购物的时候可以听到圣诞歌曲。在 IMDb，*This is Spinal Tap* 的评级突破了极限，所以他们在错误页面上故意放上一些电影中著名台词的搞笑模仿，比如"404：无法找到该页？难以置信。——Vizzinni，电影《公主新娘》"。Mailchimp 把它的吉祥物（一只穿着像邮递员的猴子）放在了几乎所有的页面上，Tumblr 的停机页面显示的是一只叫 Tumblebeast 的神奇小野兽在机房里发泄破坏，而 Twitter 的停机页面则是一只"失败鲸"（见图 3-2）。

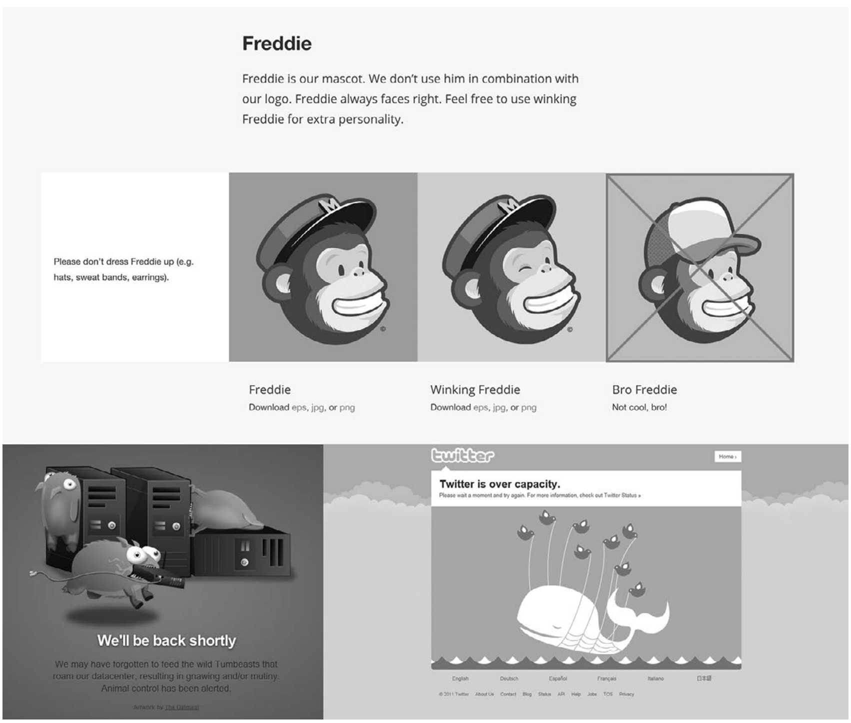


图 3-2：Mailchimp 的 Freddie（上）、Tumblr 的 Tumblebeasts（左下）、Twitter 的失败鲸（右下）

这听上去都是些小细节，但却是很重要的东西，因为设计的情感因素对用户而言就像功能因素一样重要。

> 把你的产品当作人来对待。你想让它成为什么类型的人？是有礼貌的，还是令人生畏的？是宽容的，还是严厉的？是有趣的，还是冷漠的？是认真的，还是散漫的？你想让它表现得偏执，还是令人信服？变得无所不知，还是谦虚可爱？一旦你决定了，在打造产品的时候就应该随时把这些个性特点考虑进去，使用这些个性特点作为文案、界面和功能的设计指南。无论何时对产品进行修改，都要问问自己这次修改是否符合应用的个性。你的产品是能说话的——每天 24 小时都在和你的客户交谈。
>
> ——Jason Fried、David Heinemeier Hansson、Matthew Linderman，*Getting Real*

不论你为产品选择的个性和声音是什么，我都建议你应该把“有礼貌”作为它的性格之一。如果人们把你的软件当人来看待，不妨让它懂事一点。下面列出了几个例子。

要考虑周到

> 即便我是这个程序唯一认识的人类，它也毫不在乎我，对我就像陌生人一样。
>
> —— Alan Cooper，*The Inmates Are Running The Asylum*

只要有可能，我们就应该把软件设计得像把你记在心上、考虑周到的人一样。要记住用户的参数设置，记住他们上次使用你的软件做了什么事情，记住他们过去搜索了什么东西，要尝试使用这些信息预测用户在以后会做什么事情。例如，大多数网页浏览器都会记住你过去输入的网址。Google 的 Chrome 浏览器甚至更进一步，只要你一输入 www.goo，它不仅会替你把网址补充为 Google 首页，而且如果该网址是你之前已经多次输入的，它还会在你点击回车之前就开始抓取网页，让网页加载得更快。Google 对于密码的考虑也很周到，如果你最近修改过密码，而不小心还用老密码去登录，Google 会提醒说“你的密码已经在 12 天前修改过”，而不是给你标准的“密码无效”的错误消息。

要积极响应

好的设计会响应用户的需求。例如，Apple 的笔记本电脑会检测房间中的光线强度，自动调整屏幕亮度和键盘背光。当然，响应能力也没必要做得太过花哨，经常被忽视的一种最简单的设计元素，就是要提供基本的反馈。用户点击按钮了没？要给他们提供确认点击的指示，比如改变按钮的外观或者发出声音。点击之后是不是要花时间去处理？要给用户一个指示，让他们知道处理工作正在后台执行，可以用进度条或者小窗口来提示。程序员通常都会忽视这一点，因为在本地测试是在他们自己的计算机上进行的，几乎都是瞬间就完成了。而在现实当中，这样的处理过程可能是在成千上万公里之外的繁忙的服务器上进行的，存在相当严重的滞后。如果 UI 没有给出反馈，用户就不知道点击操作是否执行了，要么会连按 10 多次按钮，要么就失去信心完全放弃了。

要宽容

人都会犯错，而且还会不断犯错。在设计软件时，要假设用户也会输入错误、点击错误的按钮或者忘了一些重要的信息。例如，在 Gmail 中发送 email 时，它会扫描你写的文字，看看是否有“附件”这个词，如果你忘了附上一些东西，它就会弹出一个确

认对话框，核实你是否是故意的。同样，在你点击了“发送”按钮之后，Gmail 会留给你几秒钟的时间，让你可以“撤销”操作，以防你改变主意或者忘记了一些重要细节。我希望所有的软件都有“撤销”按钮。有时候，我希望生活也有撤销功能。

“对错误的发生要宽容”这一点真的太重要了，我们不妨再多讨论一番。

> 这里不谈**人为的错误**，而是探讨沟通与交互：我们通常把糟糕的沟通或交互称为错误。当一个人与另一个人合作时，永远不要用错误这个词来形容另一个人的表达方式。因为每个人都希望能够理解他人的话语并做出回应，如果出现了无法理解或看似不恰当的地方，可以质疑，可以澄清，可以继续合作下去。那为什么人与机器之间的交互不能看作是合作呢？
>
> ——Don Norman，《设计心理学》

没有人喜欢错误消息，没有人想看到“PC Load Letter”这样不知所云的提示，最重要的是，没有人想感到出错是因为他们犯错导致的。在线表单通常是最让人恼火的，你花了很长时间去填写几十个文本框，点击提交，页面一重新加载，就在页面上看到了一个很难理解的错误消息。有一些网站特别让人恼怒——你根本就不知道做错了什么，输入的所有数据就都不在了，这表明设计人员并没有彻底想清楚应用程序的错误状态。以下是一些经验法则，可以避免这种错误的发生。

- 提供帮助和指引，而不是错误消息。例如避免使用“错误”“失败”“问题”“无效”和“异常”这样的词，而是向用户解释程序希望获得的输入与用户的输入之间有什么差异。
- 在用户输入的同时进行检查（而不是在页面提交之后再进行），并分别给出肯定和否定两种反馈，给出的反馈应该在用户视线附近（而不是页面的顶部）。
- 永远不要把用户做好的东西弄丢。

Twitter 的注册表单就是一个很好的例子，在你输入的同时它会给你反馈，如果输入是有效的，就显示一个绿色的打钩标记；否则就显示红色的 × 和简洁提示，说明他们想要的输入，如图 3-3 所示。例如，密码框有一个小进度条，当你输入更安全的密码时就会被填满；如果你输入的用户名已经被注册了，就可以看到可用的类似用户名的建议；如果你的 email 地址输错了，比如拼写成“jondoe@gmial.com”，就会显示一条消息“did you mean jondoe@gmail.com”。这是一种很棒的用户体验，让我们在填写表单时不会总感觉像做文书工作一样，而是更像在和一个乐于助人的人交谈，当他听不明白的时候会有礼貌地请你讲清楚一点。

除了显示有帮助的信息，我们也应该试着做出能在一开始防止错误发生的设计。在精益制造中，这种方法称为 poka-yoke，这是一个日本术语，意思是“防止错误发生”。例如，在 Stack Overflow 上输入一个新问题时，它会自动搜索相似的问题，以免你提交重复的问题。如果你的问题可能过于主观，也会自动提醒你（例如“什么是最好的 XXX”），如图 3-4 所示。

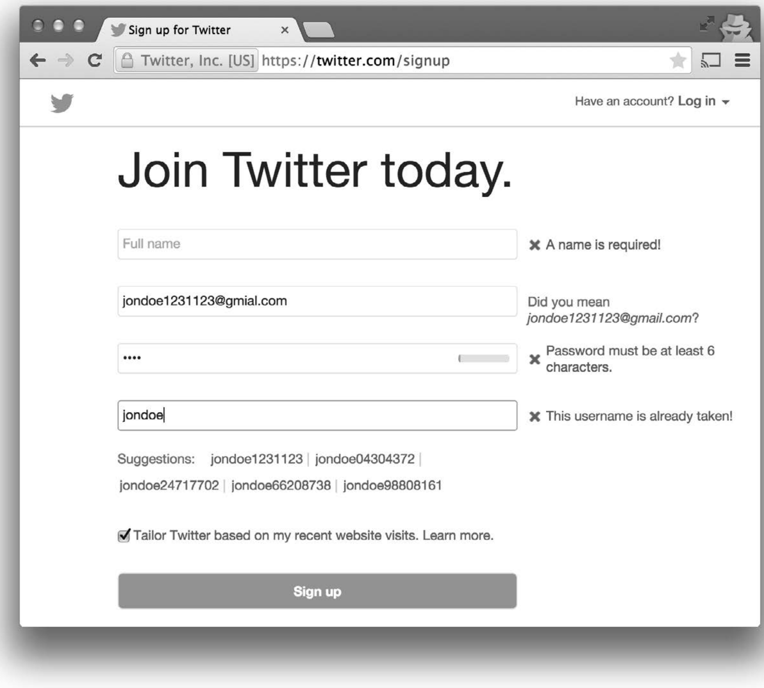


图 3-3：Twitter 的注册页面很好地呈现了帮助和指引信息

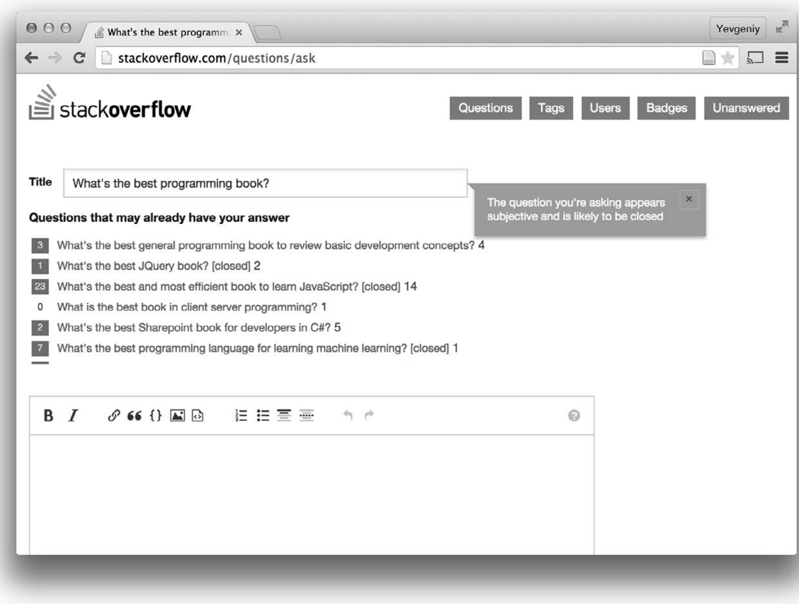


图 3-4：Stack Overflow 尝试阻止错误发生

对此有一条黄金准则，就是让错误更有可能发生或更不可能发生。例如，PC 主板被设计成每种部件都有不同类型的接口，如图 3-5 所示。这种设计保证了 CPU 不会不小心被插入到 PCI 插槽中，或者避免把网线插入 VGA 接口。现代的 ATM 吐出银行卡时，会强制你先把卡取走才可以取现金，这样你就不会忘了取自己的卡。对于软件来说，应用这一准则会更难一点，但仍然是可以做到的。例如，我在使用微软的 Word 软件的时候，还没有点击保存之前，总是当心电脑可能会崩溃。使用 Google Docs 的话，这种错误实际上就不可能发生了，因为所有的修改几乎都是立即自动保存的。还有一种更简单的做法，就是在用户点击表单的提交按钮后，立即把按钮禁用，让用户不可能重复地提交表单。

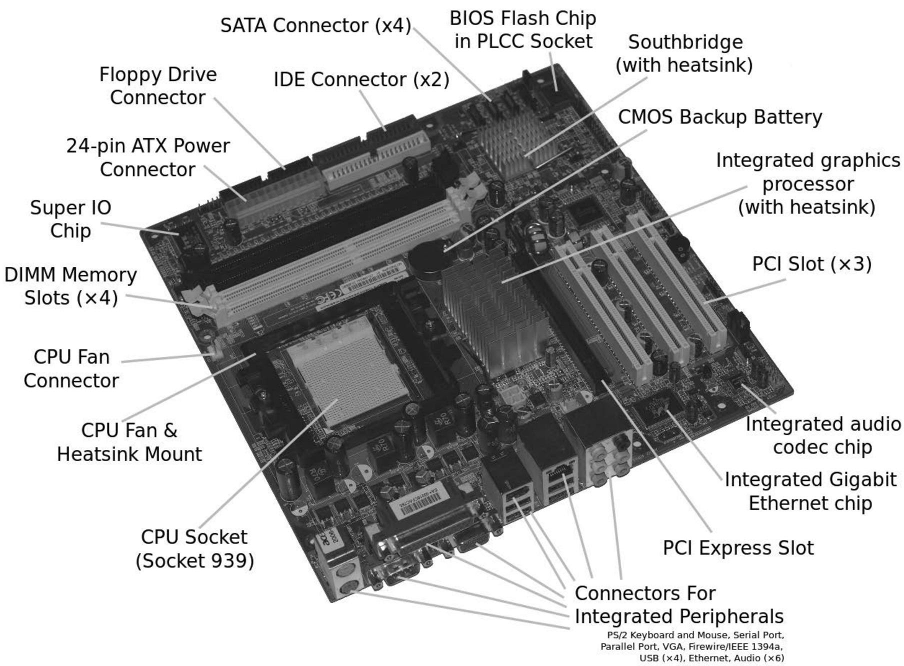


图 3-5：一些设计使得错误几乎不可能发生

除了要对错误的状态进行处理，还要确保设计能够处理**空白状态**：应用程序就像用户第一次和它交互、什么数据都没有输入的样子。用户们使用新平台联络数以百计的朋友，在动态信息中能看到朋友们的全部更新和照片，这样的平台设计也许看起来很不错。但是如果用户是第一次注册，这个设计会是什么样子呢？例如，图 3-6 展示了 Twitter 之前针对新注册用户的空白状态。



图 3-6：Twitter 以前的空白状态设计

如果动态消息是完全空白的，新用户一定不觉得是很好的体验，他们可能不会继续使用你的服务。图 3-7 展示了 Twitter 现在对空白状态的全新设计，它立即提示用户可以开始关注一些受欢迎的 Twitter 账号。

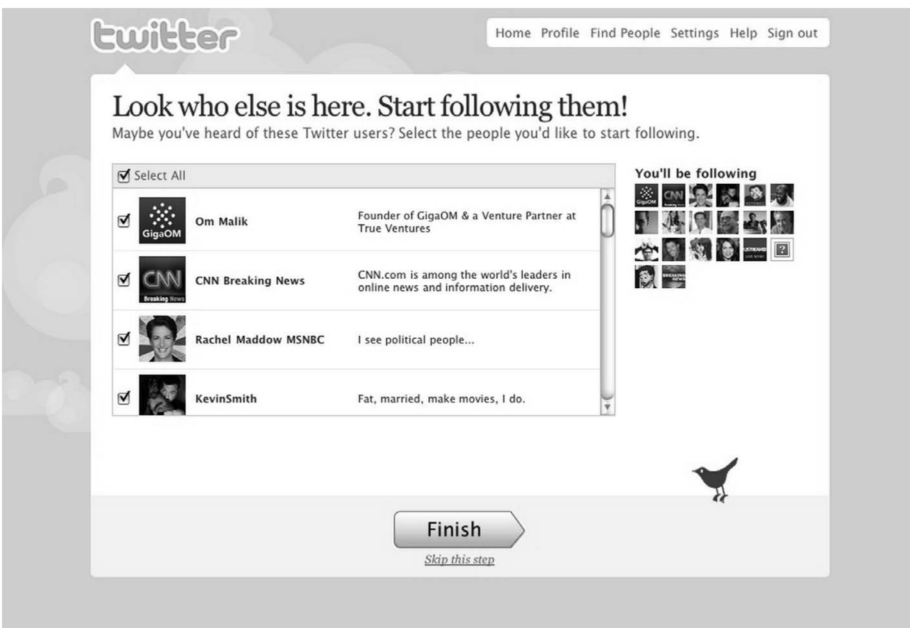


图 3-7：Twitter 全新的空白状态设计

4. 简单

几乎所有的创新训练都有一个相同的目标：简单。数学家、科学家牛顿说过：“真理往往都是存在于简单之中，而不是在纷杂与混乱之中。”《代码大全》的作者、程序员 Steve McConnell 在书中写道：“攻克复杂性是软件开发中最为重要的技术主题。”Apple 的首席设计师 Jonathan Ive 说：“简单之中蕴含着一种深刻而经久不衰的美。”每个人都在为简单而努力，问题是让事情简单并不是一件简单的事。

这种说法乍一看似乎很拗口。我们通常认为“简单”就是精简而没有多余的东西。如果从空白状态开始，只是随处添加几样东西，不就可以得到简单的设计吗？如果你写过文章或者复杂的代码，或者尝试设计过产品，就知道一开始的草稿往往都过于杂乱。我们要花大量的工作才能把这种杂乱削减成为简单的东西。

> 我本来想把信写得简短一点，但我没有时间。
>
> ——Blaise Pascal

这个问题还有一种更好的考虑方法，那就是项目其实不是从空白状态开始的，而是从大量混杂在一起的材料、知识和想法开始的。这有点像雕塑，我们拿到一块大理石，需要对它进行雕琢，日复一日，直到石头中（和脑海中）的雕像完全显露出来。正如 Antoine de Saint Exupéry 所言：“达到完美并不是没有东西可以添加，而是没有东西可以去除。”也许是去掉产品中多余的功能，也许是去掉文章中多余的文字，也许是去掉软件中多余的代码。我们可以不断去掉东西，直至剩余的都是设计中最核心的东西以及该设计和其他产品的差异，除此以外别无他物。此谓之简单。

简单其实就是一件我必须完成的事。我的产品必须完成的一件事是什么？我的设计必须向用户传达的一件事是什么？定期问问自己这几个问题，得到答案后亦可再次发问。我所设计的产品是否做了这样一件事？抑或我迷失在了细节的实现当中，产品最终做的是其他的事情。

相反的问题同等重要：我的产品**不应该**做什么？每一个额外的功能都会有特定的成本。用实物来说明能更好地表达这种成本。想象有一把瑞士军刀，里面塞了 10 种工具：小刀、螺丝刀、开罐器、小钳子，等等。现在你要考虑添加一把剪刀，剪刀会占用很大的空间，所以必须把军刀做得更大，或者把现有的工具布置得更紧凑。无论用哪种方式，都会使小刀的使用更不方便、生产成本更高。因此，要么得把刀做得非常厚才能增加新的工具，要么就得去掉原来 10 种工具中的一种以腾出空间。

软件也会面临同样的权衡取舍——每一种新的功能都会使之前的功能更难使用，令软件的生产成本更高——但这一点不是非常明显。事实上，多数公司都认为开发出更好的软件就是不断为其增加新的功能，不断发布新的版本，直到它可以做所有的事情。可惜这样是不现实的，因为根本没有人知道该怎么使用，如图 3-8 所示。

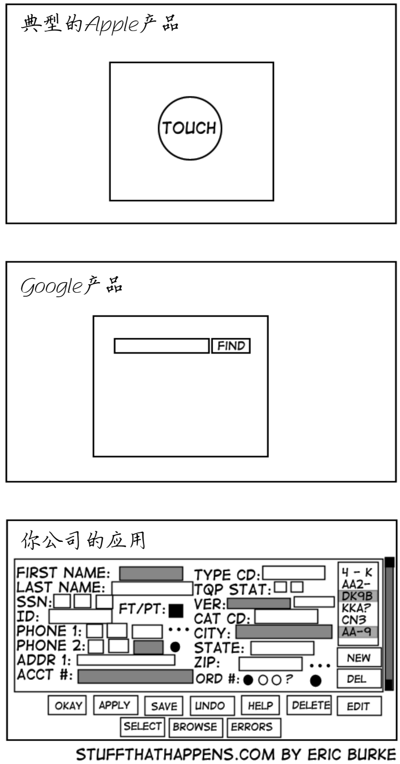


图 3-8：简单（图片由 Eric Burke 提供）

在设计上做得比较成功的公司，均认识到加入软件中功能的数量并不受“瑞士军刀空间有限”那样的物理限制，而是受使用软件的人的心理限制。设计需要简单并不是因为简单更优美，而是因为人的记忆在同一时间只能处理少数几件事。如果设计中塞入太多东西，很快就会超出人的记忆局限，用户会觉得产品功能过多而无法使用。这就是我们为什么必须限制所有设计的信息数量（更少的文本、更少的按钮、更少的设置）和所有产品的功能数量的原因（阅读 3.2.2 节了解更多信息）。

> 人们认为专注就是要对你关注的东西点头称是，但是事实并非如此。专注其实是要对其他一百个出现的好主意说不——所以你只能小心挑选。实际上，我对我们不做的事与我们做了的事一样感到自豪。创新就是对 1000 种东西说不。
>
> ——史蒂夫·乔布斯

大多数程序员最爱删代码，特别是发现有更简洁的问题解决方案时更是如此。通常来说，得对问题有更深入的理解，才能找出更优雅的实现方式。设计也是同样的道理，我们应该享受去掉功能和砍掉部分设计的过程，特别是在找到更优雅的解决方案的情况下。和编程一样，要找出更优雅的设计，需要对问题形成更深入的理解。

我们有时可以通过用户研究、客户开发和“五个为什么”这样的技巧来培养自己的这种理解力。但现实中很多人关注的问题都属于**棘手的问题**——还没等真正理解问题之所在，就必须**先**找出解决方案。也就是说，解决问题可以让我们对问题有更清晰的见解，这样又可

以找出更胜一筹的解决方案。形成了新的解决方案后，又可以对问题有更好的理解，如此重复循环。设计就是不断迭代，只有进行多次迭代，才可能找到困难问题的简单解决方案。

实际上，让解决方案简单并不是目标。用 Apple 首席设计 Johnthan Ive 的话说，真正的目标是把问题解决到“人们根本没有察觉到解决方案的存在，（并且）也没有觉察最终解决的这个问题有多难”。最要紧的并不是解决方案是简单的，而是让用户的生活变得简单。iPhone 的设计就是一项超乎人们想象的复杂技术，但是它的使用却很简单。而检验是否成功让用户的生活变简单的唯一方法，就是在用户使用你的产品时观察他们，这个过程的正式名称叫作**可用性测试**。

5. 可用性测试

上一章介绍过一个概念，不管我们进行了多少思考和验证，有一些设想仍然是错误的。而解决方法就是把产品放到真正的客户面前，测试你的那些设想（阅读 2.2.2 节了解更多信息）。同样的逻辑也可以应用到设计上面，不管你多么擅长以用户为中心的设计，有一些设计理念仍然是不起作用的，把它们找出来的唯一方法就是让设计通过可用性测试呈现在真正的用户面前。

不要把可用性测试与焦点小组[2]（focus groups）混为一谈。焦点小组的目标是了解人们如何**看待**某个点子或某种产品，而可用性测试的目标则是了解人们如何**使用**你的实际产品去完成特定的任务。虽然有一些公司可以帮助你开展正式的可用性研究，但一般都昂贵且费时，大多数创业公司都可以用更简单的方法去实现。下面列出了大概的步骤（读者可阅读《点石成金》一书了解更完整的介绍）。

(1) 把少数用户（3~5 个）带到你的办公室。
(2) 准备好录像设备（例如安装在三脚架上的 iPhone）。
(3) 记录下用户使用你的产品执行一系列任务的过程。
(4) 让团队成员观看录像。
(5) 根据你们认识到的情况决定采取什么行动。
(6) 每 3~4 周重复一次。

如果之前从未进行过可用性测试，你很快就知道第一次观察公司以外的人使用你的产品会是一种发人深省的体验。每个月只需要花几个小时，就会定期掌握产品设计的有关情况，这是其他任何方式都做不到的。要记住最重要的一点是，如果你作为协助者和用户一起待在房间里，你就在那观察就行，不要去干扰用户。你可以鼓励用户，回答一些后勤方面的问题，但是不要**帮他们使用产品**，特别是在他们犯了错误的时候。用户可能会感到挫折，但是可用性测试的全部意义就在于找出这些错误和挫折，然后去解决它们。

除了可用性测试，还有其他几种手段也可以用来改进设计。有一种方法就是直接在产品中加入某种机制，让它可以方便地发送反馈，比如在网页上放一张反馈表单。虽然只有很少部分的用户会花时间发送反馈给你，但如果他们做了，通常都是一些很有价值的内容。第二种方法就是定期开展可用性调查，类似于把反馈表单直接发送到每个用户的收件箱。正确地开展可用性调查是一门艺术，我们可以搜索专门的可用性产品，用来处理一些细节问题（例如 survey.io）。

注 2：焦点小组也称焦点团体、焦点群众，是质性研究的一种方法，就某一产品、服务、概念、广告和设计，通过询问和面谈的方式采访一个群体以获取其观点和评价。——译者注

3.1.3　视觉设计

现在把关注点放到设计的视觉方面。数千年以来，人们一直都在进行着视觉设计，所以这是一个非常深奥的领域。这一节也只不过是视觉设计的“Hello, World”式教程。开始学习一门新的编程语言时，首要目标一般都是学习如何用这门语言创建一个程序，把“Hello, World”输出到屏幕上，在深入研习如何实现更复杂的程序之前，快速运行一些简单的东西，帮助我们建立自信。同样，在下面这份教程中，我的目标也是向大家介绍些基本的设计技能，以便让一些简单的东西发挥作用，让你先树立信心，再去更深入地研究和学习如何实现更复杂的设计。

这些基本的视觉设计技能和技术分别是：

- 文案；
- 设计重用；
- 布局；
- 排版；
- 对比与重复；
- 颜色。

在这份教程中，我将主要关注两个例子：一是解决一份简历存在的设计问题，这是几乎所有人都非常熟悉的设计任务；二是从头开始设计一个网站（具体而言就是本书的配套网站 hello-startup.net），其过程有点像典型的创业产品所需要经历的过程。希望大家不要把注意力放在我为简历和 hello-startup.net 这两个设计所做的具体的设计决定上，因为这些决定并不是万灵药，而是重点关注我做这些决定的思考过程。

1. 文案

尽管许多人都认为颜色、边框、图片和花哨的动画就是设计的主要手段，但实际上所有软件设计的真正核心几乎都是文案。事实上，如果去掉大多数应用程序的颜色、边框、图片和其他一些东西，只留下文本，该设计很可能依然是最低限度可用的。这并不是说这些元素就无关紧要，而是说用户在软件产品中需要的绝大部分信息就在标题、头部、正文、菜单和链接中，所以即便进行视觉设计，最先考虑的事情永远都是文案。

> 伟大的界面是写出来的。如果你认为每一个像素、每一个图标和每一种字体都很重要，那么你也要相信，每一个字母都很重要。
>
> ——Jason Fried、David Heinemeier Hansson、Matthew Linderman，*Getting Real*

花点时间彻底想清楚，你要告诉用户什么以及如何告诉他（阅读 3.1.2 节了解更多信息）。出色的标题和内容提要（即所谓的电梯陈述[3]）都是特别重要的，因为当用户第一次使用你的应用程序，在搜索结果中看到你的应用程序，或投资者听你陈诉点子时，这些内容是他们第一眼看到的东西。不知道你有没有注意过，当你翻阅杂志、报纸或者科技期刊的时候，你只会阅读其中的一些文章？你有没有停下来考虑过，为什么会阅读这些文章而不是其他文章？你的大标题必须能够和目标人群的角色产生共鸣，不仅告诉他们你要做什么（“我们的软件可以实现 XXX”），还要告诉用户为什么应该关注它（“我们的

注 3：是指陈述者应该能在乘电梯的时间内，也就是在 30 秒 ~2 分钟之内完成自我介绍。——译者注

软件可以实现 XXX，所以你可以成功地 YYY”）。知道如何提炼出清晰的信息去介绍你的动因、你的使命，是各种事情成功的关键因素之一（阅读 4.2.3 节和 9.2.1 节了解更多信息）。

我们通常都会犯这样一个错误，就是把文案留到最后再考虑——或是更糟，在设计时只放一些占位文本，比如标准的拉丁语填充文本 lorem ipsum。这样会把文案这个设计的最重要部分弱化为仅是文本的形状，看不到文本随实际数据而发生的变化，也不注重写出能与受众产生共鸣的信息。我在搭建本书网页时做的第一件事，就是把麦克、莫妮卡和马赫什想看到的信息写下来（阅读 3.1.2 节了解更多信息），如图 3-9 所示。我先大概写一些基本的框架——有关本书的信息、作者、购买图书的方法、最新消息和创业资源——然后再把内容细节填进去，最终得到了大量简洁、语义化的 HTML 代码。结果虽然没有什么特别的吸引力，但是要知道设计是迭代的，这仅仅是最初的草稿。

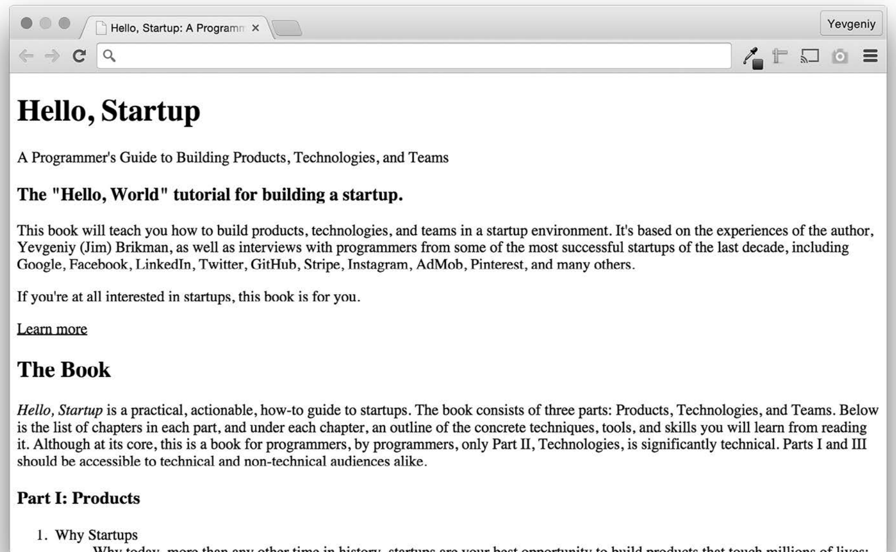
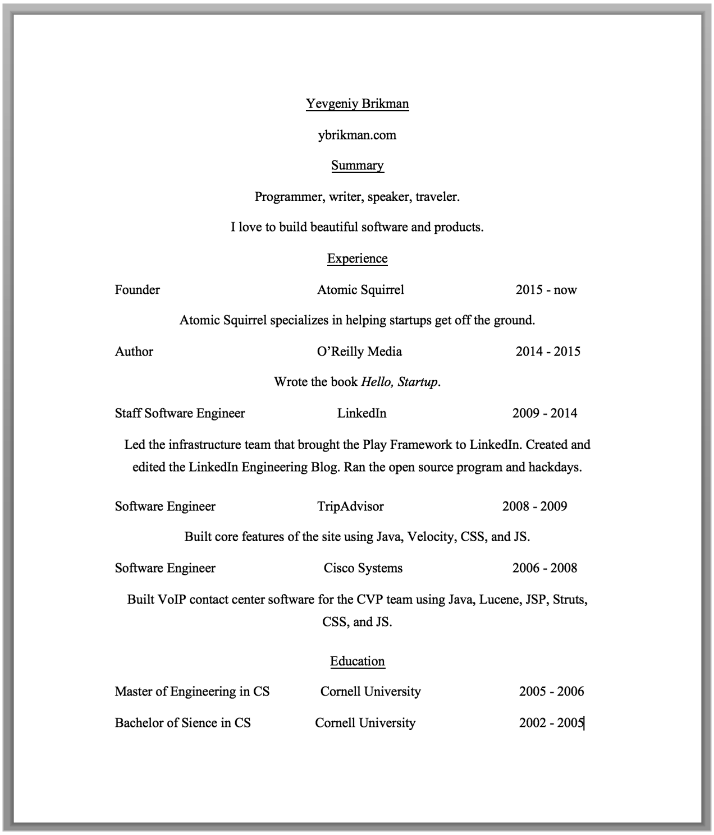


图 3-9：hello-startup.net 的文案

简历最初的草稿如图 3-10 所示，这差不多就是我在过去几年看到的数百份简历的风格。看起来有点儿丑，但是相关的文字材料已经放在适当的位置上，所以是一个很好的起点。



Yevgeniy Brikman

ybrikman.com

Summary

Programmer, writer, speaker, traveler.

I love to build beautiful software and products.

Experience

Founder	Atomic Squirrel	2015 - now

Atomic Squirrel specializes in helping startups get off the ground.

Author	O'Reilly Media	2014 - 2015

Wrote the book *Hello, Startup*.

Staff Software Engineer	LinkedIn	2009 - 2014

Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer	TripAdvisor	2008 - 2009

Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer	Cisco Systems	2006 - 2008

Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

Education

Master of Engineering in CS	Cornell University	2005 - 2006
Bachelor of Sience in CS	Cornell University	2002 - 2005

图 3-10：一份存在诸多设计问题的简历

2. 设计重用

好的艺术家模仿，伟大的艺术家偷窃。

——史蒂夫·乔布斯

如果你刚接触设计，或者几乎没有经过任何专业训练，一开始最好的方法就是模仿他人的作品。不要重新发明车轮，如果你不是造车轮的专家的话更是如此。事实上，即便你是造车轮的专家（即经验丰富的设计师），仍然应尽可能地重用现有的设计（同样的道理也可以用在代码重用上，5.3 节将介绍）。模仿他人既节省了时间，也是在学习，还可以获得高质量、经得起考验的作品。在学习设计时，复制与粘贴似乎不是一种很让人满意的方法，但上一章讨论过，模仿、转换与合并是所有创造性工作的基本组成（阅读 2.1 节了解更多信息）。

我开始每一个项目的时候，都是先浏览现有的设计，了解有哪些可以重用或哪些是自己需要的。例如，现在有无数的网页、移动应用界面和 email 模板可供使用，我们不一定要从头去构想设计作品。我的最爱之一就是 Bootstrap，它不仅是一个模板，而且还是一个开源、响应式的 HTML/CSS/JavaScript 框架，搭载一整套默认的样式、行为、插件和可重用的组件。

如果你不想立马跳到代码中，可以先使用线框图或原型工具，比如 Balsamiq、UXPin 或者 Justinmind。利用这样的工具，可以从 UI 元素库拖拽出一些元素进行摆放，组合成一份设计。现在也有数以百计的网站，可以在上面找到照片、图形和字体，其中有一些是免费的（例如 Wikimedia Commons 和 Google Fonts），有一些则是付费的（例如 iStock 和 Adobe Typekit）。最后，还可以利用一些设计社区，比如 Dribbble（设计师可以在这个网站上分享和讨论他们的作品）和 DesignCrowd（这是一个网上市场，能快速找到自由职业者帮你设计 logo 或网站）这样的网站。读者可访问 http://www.hello-startup.net/resources/design 和 http://www.hello-startup.net/resources/images-photos-graphics 查看完整的设计资源列表。

hello-startup.net 的最终设计大体上就是根据一个叫 Agency 的免费 Bootstrap 模板设计而成的，而简历的最终设计则是以 Hloom 上的一个模板为基础的。但为了帮助大家训练出设计师的眼光，我不会马上就使用这些模板，而是一步一步地做出来，让大家学着辨别视觉设计的不同方面。先从**布局**开始。

3. 布局

在一些做得比较好的布局中，我们可以根据界面上元素的位置推断出许多信息。布局有一个要点就是**亲密性**，元素之间的亲近程度表明它们在逻辑上是否相关联。逻辑上联系在一起的元素应该更加接近，没有关联的元素则应该远离一些。看看图 3-11，左边是原来的简历，右边虽然是同一份简历，但是对亲密性的处理更好一点。

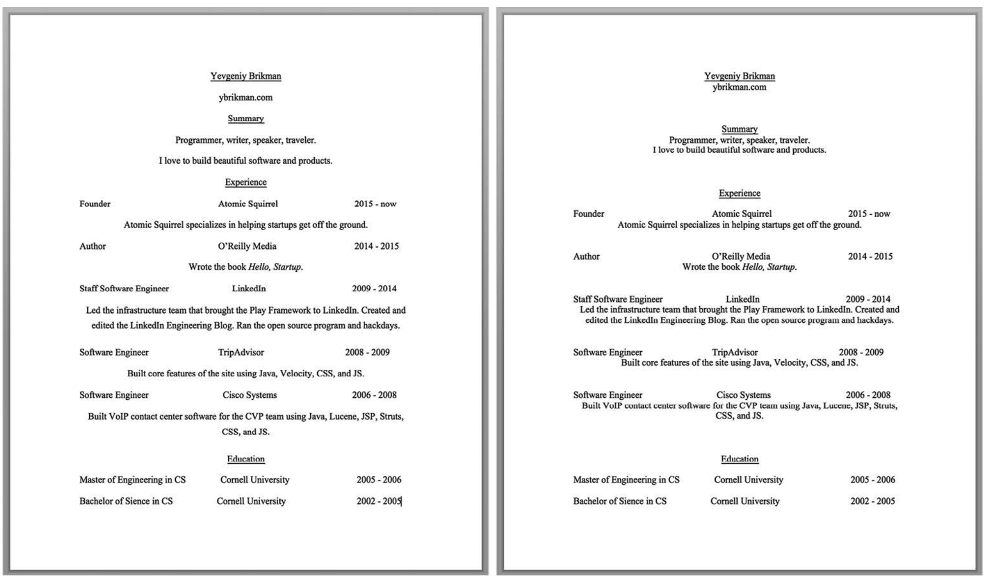


图 3-11：左侧是原来的简历，右侧是同一份简历，但对亲密性的处理更好

我在不同部分（简介、经历、教育）之间都放入两个新行，每一节的标题和其内容之间只放入一个新行（例如 Summary 和 Programmer，writer，speaker，traveler 之间）。然后，把每份工作的信息放得更接近一些，但是在不同的工作之间加入一个新行，这样一份工作的开始和另一份的结束就能区分开了。我们对 hello-startup.net 的设计进行了类似的处理，如图 3-12 所示。

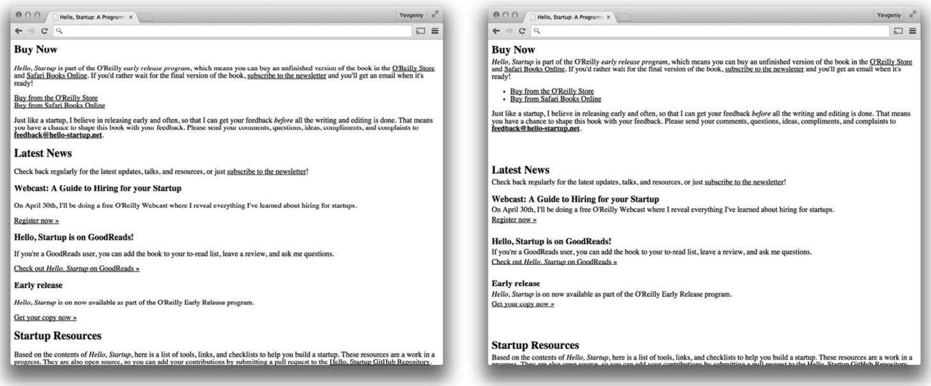


图 3-12：左侧是 hello-startup.net 原来的设计，右侧是同一份设计，但对亲密性的处理更好

右侧的设计能更明显地看出 Buy Now、Lastest News 和 Startup Resources 是不同的部分，因为它们之间的间隔增加了。从中还可以看到 Webcast: A Guide to Hiring for your Startup 与下面的两行在逻辑上是一个整体，因为它们之间的间隔小了。

我们要努力平衡好关联元素之间的亲近和不关联元素之间的大量空白。人的大脑在同一时间能处理的信息量是有限的，保持可读性的一个关键因素就是在元素之间要保留许多空白，这样才能让人们一次仅关注一样东西。这里我强调的是要加入许多空白，大部分初学者总是尝试把所有东西都紧紧地挤在一起，所以好的经验法则就是“使用双倍的空白”：每一行之间都加入间隔，每个元素之间都加入间隔，每组元素之间都加入间隔。Medium 是一个以精美设计而闻名的博客平台，它有一个启发灵感的例子，让我们知道单纯利用空白与排版可以做出多么出色的效果，如图 3-13 所示。

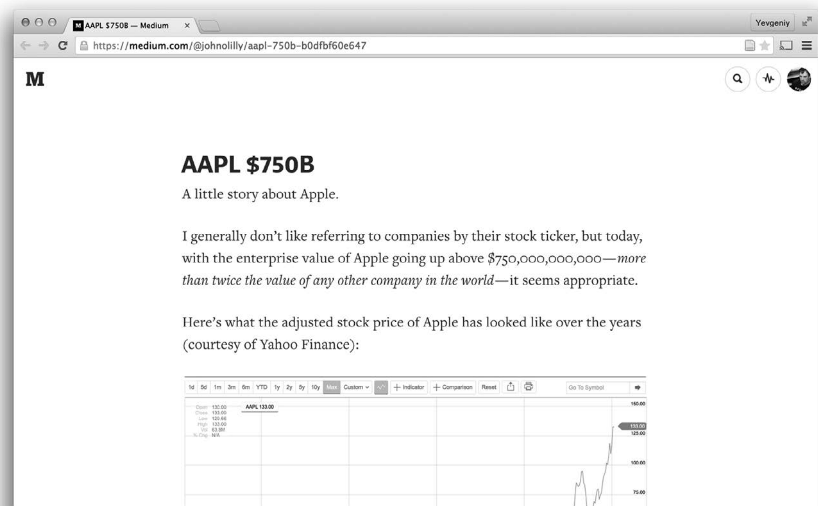
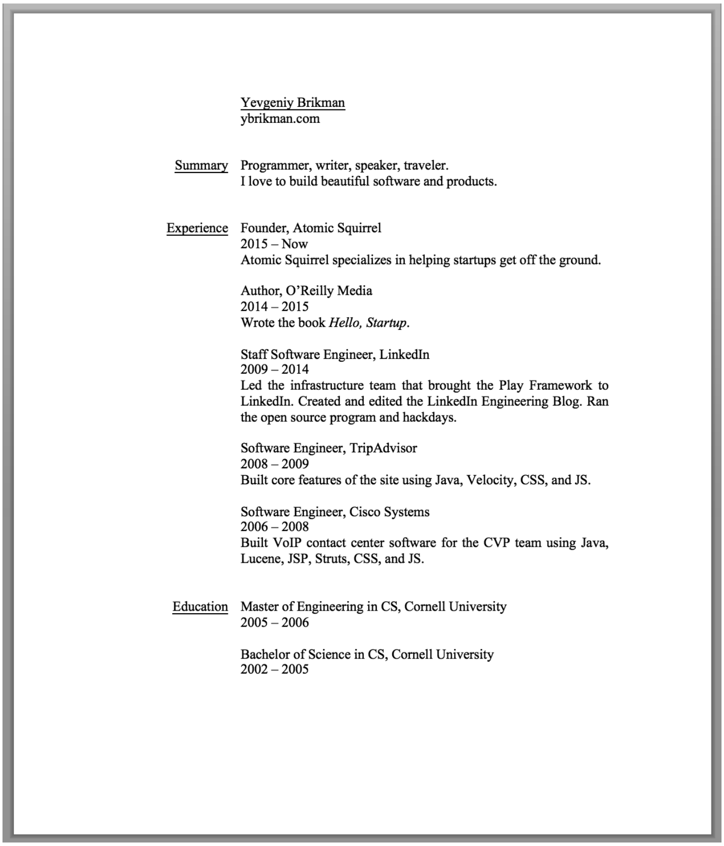


图 3-13：Medium 炫耀他们对空白的利用

布局的另一个关键因素是**对齐**。对齐可以表达元素之间存在一种关系，且不需要让它们更接近或者离得更远（即实现亲近的要求），只要把它们沿着共同的线排放即可。以下是对齐的黄金规则。

> 无论什么都不应该在页面上随便摆放。每一个元素都应该和页面上的另一个元素有某种视觉上的关联。
>
> ——Robin Williams，《写给大家看的设计书》[4]

请注意观察图 3-11 的简历，它有许多不同、看似随意的对齐，比如：每部分的标题是居中对齐，工作标题是左对齐，公司名称是居中对齐（但不是太美观，只使用空格和制表符对齐），日期则是右对齐（同样不美观），工作描述则是居中对齐。图 3-14 展示的是同一份简历，但使用了统一且更明显的对齐方式。



Yevgeniy Brikman
ybrikman.com

Summary
Programmer, writer, speaker, traveler.
I love to build beautiful software and products.

Experience
Founder, Atomic Squirrel
2015 – Now
Atomic Squirrel specializes in helping startups get off the ground.

Author, O'Reilly Media
2014 – 2015
Wrote the book *Hello, Startup*.

Staff Software Engineer, LinkedIn
2009 – 2014
Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer, TripAdvisor
2008 – 2009
Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer, Cisco Systems
2006 – 2008
Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

Education
Master of Engineering in CS, Cornell University
2005 – 2006

Bachelor of Science in CS, Cornell University
2002 – 2005

图 3-14：同一份简历，但应用了更明显的对齐

这种布局更容易阅读，因为所有元素都沿着每一节标题和每一节内容之间的一条明显的直线排放。当然，那里实际并没有线，但我们的大脑会在那里插入一条线，如图 3-15 所示。

注 4：超级畅销书，适合各行业与文字打交道的读者，中文版由人民邮电出版社出版，www.ituring.com.cn/book/1757。——编者注

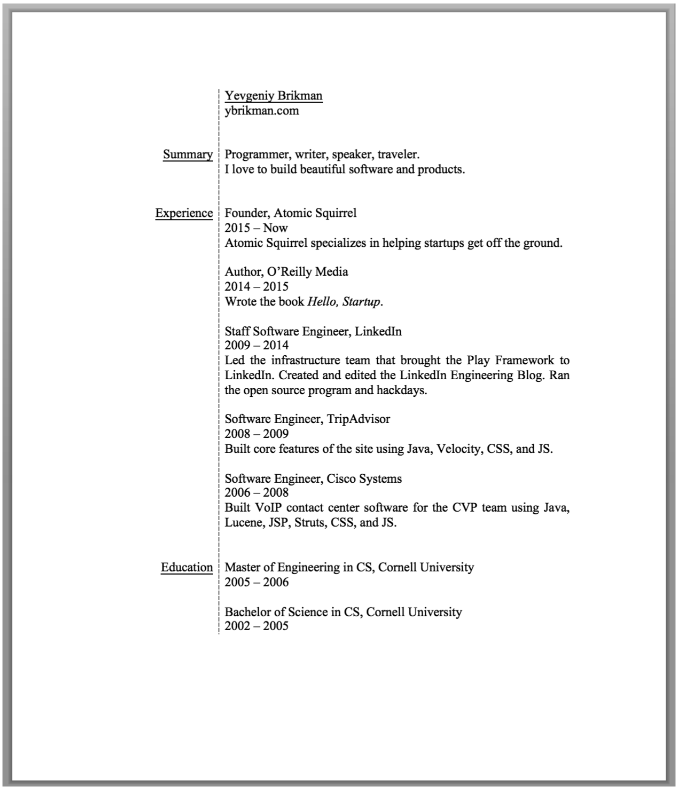


图 3-15：大脑会插入一条虚拟的线，帮助我们理解布局

这样的线在设计中随处可见，所以要让自己学会有意识地注意它们的存在。例如，图 3-16 展示了更好地使用了对齐之后的 hello-startup.net。看看这个设计中的线在哪里？

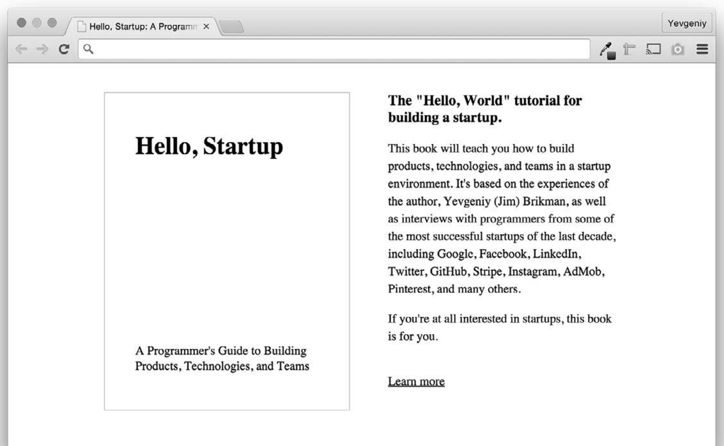


图 3-16：hello-startup.net 在设计上更好地利用了对齐

作为经验法则，我们要尽量挑选一条明显的线，让所有东西都向它对齐。换句话说，不要把一些内容左对齐，而另一些内容右对齐，有些内容又居中对齐。另外，要谨慎使用居中对齐，因为这种方式在我们的大脑中并没有建立起一条明显的线，会让设计看起来更加业余。这也仅仅是一条经验法则，你当然可以偶尔打破它，但也必须是有意识而为之。

4. 排版

排版就是安排文本的艺术与科学，目的是让文本易读和美观。这一节将关注排版最重要的几个因素：行宽、行距、字体和样式。

行宽是指每一行的长度。如果文本行太短，读者就会太过频繁地被打断而跳到下一行。如果文本行太长，读者就没耐心把它读完。我们看看图 3-17 中 hello-startup.net 的哪一个版本更容易阅读。

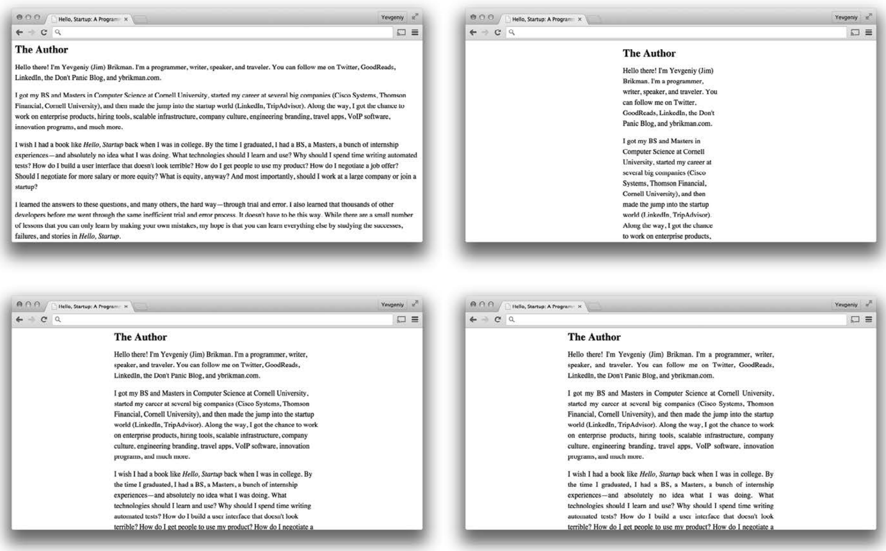


图 3-17：hello-startup.net 各种行宽的对比：左上图是 140 个字符，右上图是 35 个字符，左下图是 70 个字符，右下图是 70 个字符并且两端对齐

大多数人会觉得下面的图比上面的图更容易阅读，而总的来说右下图的效果是最好的。首先，右下图中的两端对齐在每个段落的两侧都形成了明显的线，对于大量文本的阅读是很有帮助的（这就是多数图书和报纸都使用这种样式的原因）。其次，下面的图使用了合适的行宽，每行大概就是 45~90 个字符。作为经验规则，设置行宽时只要让它足以连续容纳字母表中所有字母的 2~3 倍就足够了：

abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz abcdefghijklm

行距是行与行之间的垂直间距。和行宽一样，如果行距设置得太小或太大，文本的阅读都会比较困难。行距的最佳尺寸一般都是字体尺寸的 120%~145%，如图 3-18 的下图所示。

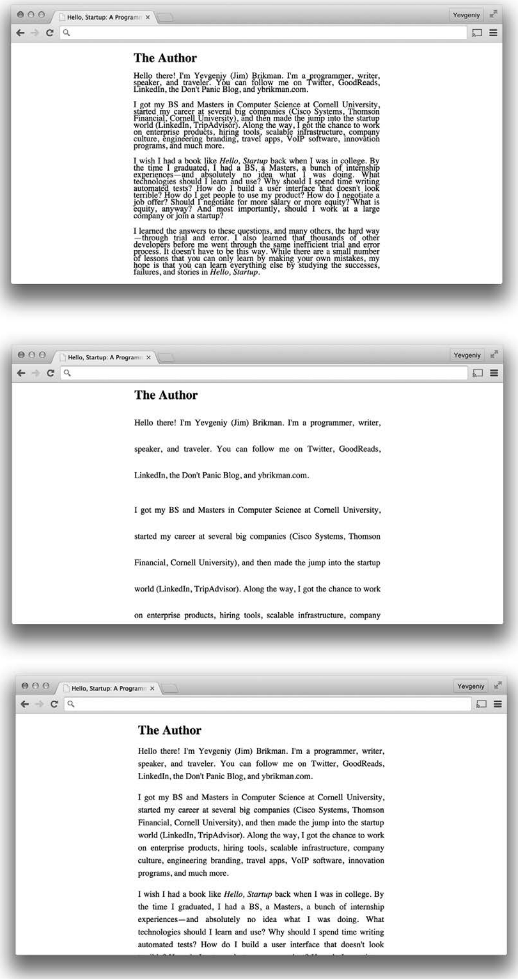


图 3-18：hello-startup.net 在字体尺寸为 16px 时设置不同行距的对比：上图行高为 13px，中图行高为 50px，下图行高为 24px

字型（typeface）就是字母的设计。每种操作系统都会预装一些标准、内置的字型，比如 Arial、Georgia、Times New Roman 和 Verdana。这些字型很多都不是特别好看，甚至被过度使用了。把它们用在大多数设计上都会很乏味。所以，显著提高设计的一个最简单方法就是不要使用系统的字型。可以从一些网站上找到高质量的替代字型，有像 Google Fonts 这样的免费网站，也有像 Adobe Typekit 这样的付费网站，但怎么知道该用上千种字型中的哪一种呢？

总的来看，所有字型都可以分成五类：衬线字型、无衬线字型、装饰性字型、手写型和等宽字型。每种分类都有大量不同的字型，有些字型放在某个分类中也未必很合适，但还是有一些经验法则可以帮到我们。

衬线字型在每个字母或符号的笔划结尾处都有一些称为衬线的细线，如图 3-19 所示。可

以看到，在单词 Serif 的 r 字母的底部，会向两侧延长出小线条，字母就像放在基座上。衬线字型的笔划在字母的不同部分的粗细通常都是不一样的。例如图 3-19 中 Serif 中的 S，上下都比中间要细。衬线和粗细的变化令每个字母看上去都更明显，有助于提高阅读速度，特别是有大量文本的时候。因此，衬线字型比较适合字数比较多的正文文本和打印材料（大多数图书正文使用的都是衬线字型）。衬线字型作为最古老的字型，不仅可以追溯到使用印刷术的年代，甚至可以一路追溯到古罗马人刻在石头上的字母。所以，如果想要有“传统的”感觉，就可以在标题中使用衬线字型。



图 3-19：衬线（Serif）字型

Sans 是一个法语单词，意思是“没有”，所以 Sans serif 字型就是没有衬线的字型。仔细观察图 3-20 中 serif 中的字母 r，底部并没有向外伸出任何线条。无衬线字型的整个字母拥有更一致的笔画粗细。例如图 3-20 中 Sans 中的 S 的各个位置的粗细都是一样的。因为无衬线字型的外观更为简单、统一，如果大量正文文本使用的是中等字号，无衬线字型的效果就不如衬线字型，但是无衬线字型通常在非常大或非常小的字号上有更好的表现，比如大标题或者小的帮助文本。事实上，如果字母太小，或者在低分辨率的屏幕上查看，衬线字型的微小细节看起来会变得模糊，所以无衬线字型在数字媒介上非常流行。



图 3-20：无衬线字型

正如其名，装饰性字型用于装饰或者强调。这些字型是独特、有趣且丰富多样的。如果想让一些文本与众不同，用这样的字型就再好不过了，如图 3-21 所示。但是，这样的字型其实并不方便阅读，所以通常会把它限制在标题或子标题的少数单词中。



图 3-21：装饰性字型

手写型看起来就像用手写的潦草字体或书法字型，如图 3-22 所示。和装饰性字型一样，使用这种字型也可以很好地起到强调作用，但只用在不多的几个单词或字母上就行了，因为手写型也是不容易阅读的。



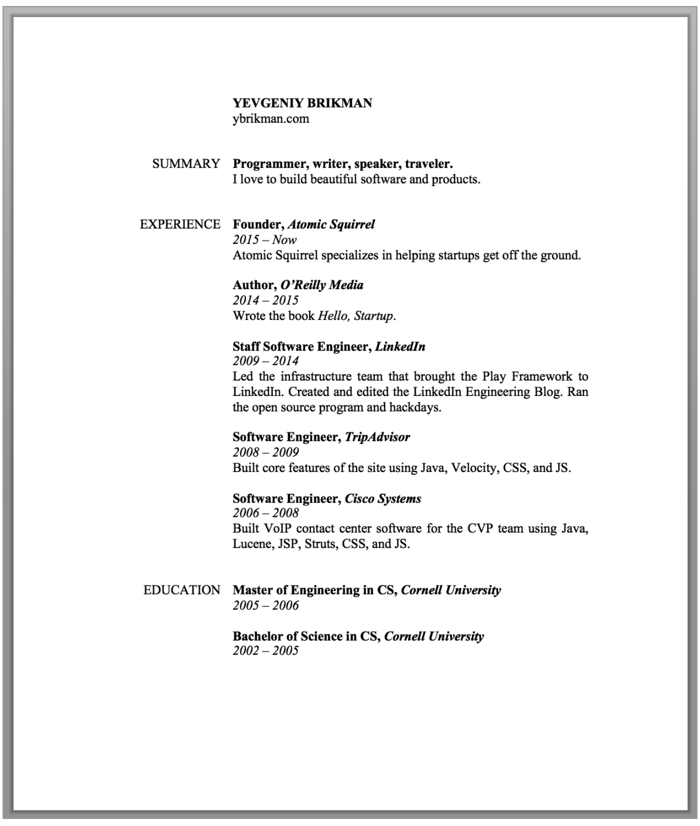
图 3-22：手写型

如图 3-23 所示，等宽字型的每一个字母都会占据同样的空间，所以通常只会在显示代码段（这就是为什么所有的终端、文本编辑器和 IDE 都会使用等宽字型）时，或者想让文本看起来像是从打字机打出来时使用。



图 3-23：等宽字型

我们可以为某种字型应用不同的**样式**，从而改变它的外观，包括文本尺寸、文本粗细（例如加粗或变细）、文本倾斜度（例如斜体）、字符间距、下划线和字母大写化。字型和样式的某种特定组合就称为**字体**（font）。设计中的每一种字体都应该服务于某个特定目的。上面简历的例子就违背了这样的原则，因为它所有的文本使用的都是 12pt 大小的 Times New Roman 字体。唯一的例外就是在几个地方用了下划线去强调每一节的标题，但是下划线并不是一种很好的选择。事实上，没有图书、杂志或报纸会使用下划线，因为这么做会让文本更难阅读。唯一的例外就是网站，下划线可以用来表示超链接。所以我们不应该在其他地方使用下划线，以免引起混淆。因此，去掉简历中的下划线，使用几种字体样式来增强设计的效果，如图 3-24 所示。



YEVGENIY BRIKMAN
ybrikman.com

SUMMARY **Programmer, writer, speaker, traveler.**
I love to build beautiful software and products.

EXPERIENCE **Founder, *Atomic Squirrel***
2015 – Now
Atomic Squirrel specializes in helping startups get off the ground.

Author, *O'Reilly Media*
2014 – 2015
Wrote the book *Hello, Startup.*

Staff Software Engineer, *LinkedIn*
2009 – 2014
Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer, *TripAdvisor*
2008 – 2009
Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer, *Cisco Systems*
2006 – 2008
Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

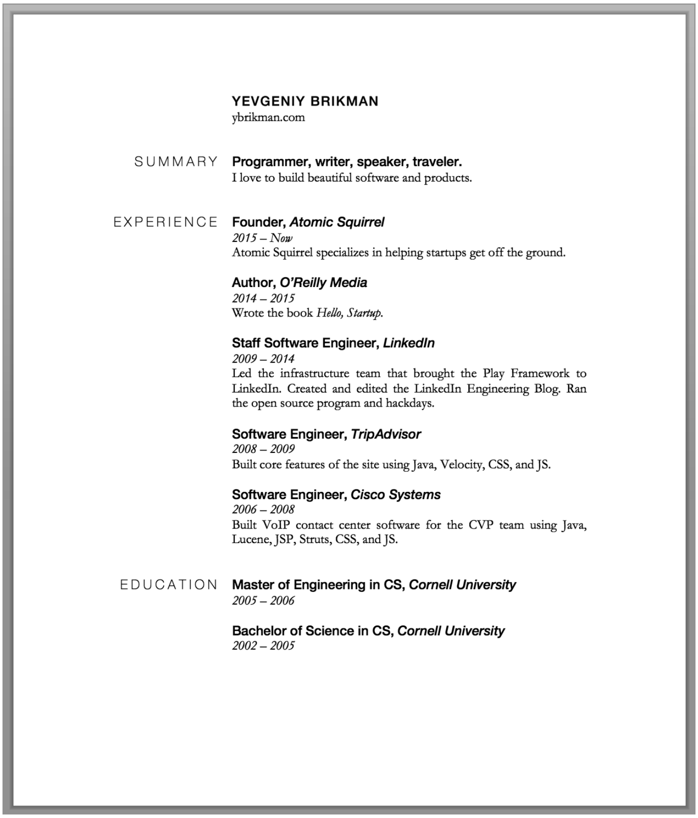
EDUCATION **Master of Engineering in CS, *Cornell University***
2005 – 2006

Bachelor of Science in CS, *Cornell University*
2002 – 2005

图 3-24：使用几种字体的简历

现在简历的结构更加清晰了：所有工作和教育经历的标题都是加粗的，所有的公司和学校名称都是加粗和斜体，所有的日期都是斜体，每一节标题都是大写字母。这是一种改进，但看起来仍然有点儿乏味，因为整份简历只使用了一种字型——Times New Roman。

我们可以通过一些试验和经验积累找出哪几种字型放在一起会更好看。如果你刚接触字体，可能需要让专业人士去处理。如果搜索“字体搭配”，可以找到几十个可以提供提前验证过的精美推荐字体组合的网站，例如 Google Web Fonts Typographic Project（Google 网页字体排版项目）展示了 Google 字体的数十种可行的组合方式，Just My Type 关注的是 Adobe Typekit 字体的组合，而 Fonts in Use 则有一些实际运用的精美排版，并可以根据行业、格式和字型进行筛选。我在 Fonts in User 中找到了很多可以用在简历上的优秀方案，但我挑了一种比较保守、在别人的电脑上也可以起作用的方案，即标题选用 Helvetica Neue，正文选用 Garamond，如图 3-25 所示。



YEVGENIY BRIKMAN
ybrikman.com

SUMMARY **Programmer, writer, speaker, traveler.**
I love to build beautiful software and products.

EXPERIENCE **Founder, *Atomic Squirrel***
2015 – Now
Atomic Squirrel specializes in helping startups get off the ground.

Author, *O'Reilly Media*
2014 – 2015
Wrote the book *Hello, Startup*.

Staff Software Engineer, *LinkedIn*
2009 – 2014
Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer, *TripAdvisor*
2008 – 2009
Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer, *Cisco Systems*
2006 – 2008
Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

EDUCATION **Master of Engineering in CS, *Cornell University***
2005 – 2006

Bachelor of Science in CS, *Cornell University*
2002 – 2005

图 3-25：使用多种字型的简历

针对 hello-startup.net，我使用了 Agency 模板的字体，分别是 Monsterrat、Droid Serif 和 Roboto Slab（均可以在 Google Fonts 中免费下载），如图 3-26 所示。

这些新字体使设计看起来更加简洁明了。但总体来看，仍然是相当单调的。我们需要添加一些对比，让设计更加活跃。

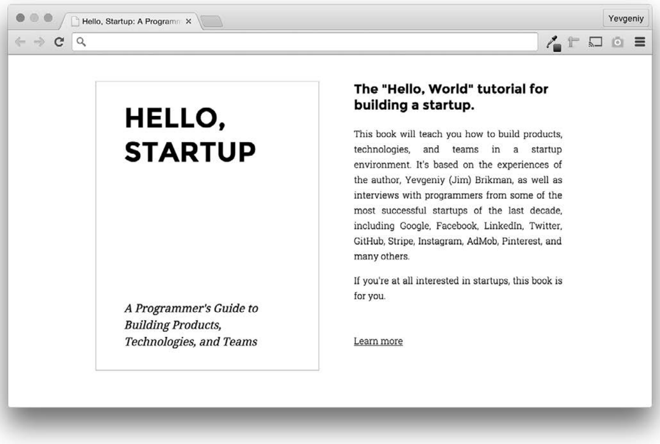


图 3-26：使用 Montserrat、Droid Serif 和 Roboto Slab 字型的 hello-startup.net 页面

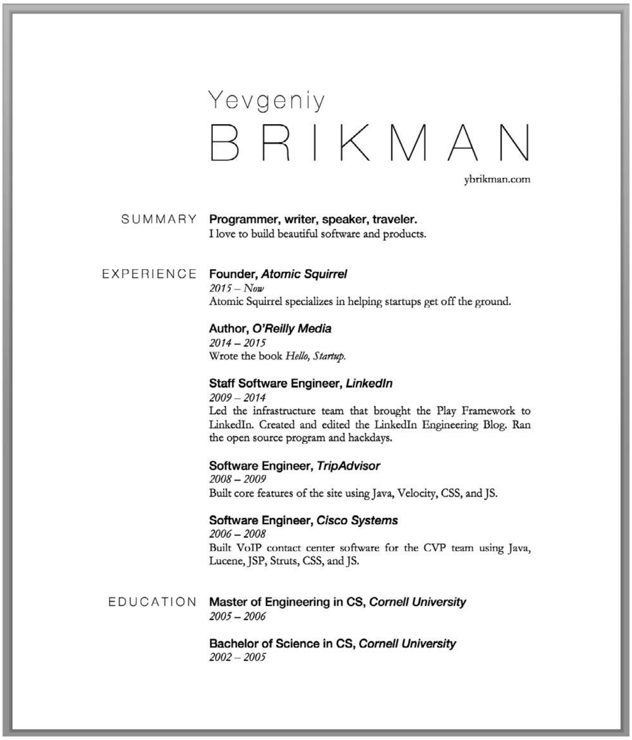
5. 对比与重复

亲密性与对齐可以表示两个元素是有关联的，而**对比**则可以明显地区分设计中的两个部分。例如，当我们把多种字体混搭在一起的时候，有一件最重要的事情需要弄清楚，就是必须让它们之间形成强烈的**对比**。在前面的简历里，工作职位是加粗的 Helvetica Neue 字体，这样就可以很轻松地把职位从工作描述（用了普通的 Garamond 字体）中区分出来。请读者注意观察这种信息是如何通过**重复**得到增强的：所有的工作职位都使用同一种字体，每一部分的标题都使用另一种字体，而所有的内容文本都使用第三种字体。只要明确了某种设计元素的目的，不论是所选择的字体，还是角落的 logo，或者是元素对齐的方式，都应该在所有地方进行重复。这种重复可以形成你的品牌（阅读 4.2.4 节了解更多信息），如果这种重复足够明显，读者在任何地方都可以识别出你的风格（见图 3-27）。



图 3-27：让同一种风格重复出现，打造自己的品牌

我们也可以通过样式的变换（例如字号大小、加粗、字母大写）和字型的选用，让字体形成对比。如果选用的两种字体太过相似，比如 12pt 和 14pt 的同一种字型，或者使用衬线字型的两种字体，这些字体就会引起冲突，设计看起来就会有问题。因此，每当选用新字体的时候，都必须是为了某个具体的目标。而为了强化这一目标，就必须让字体体现强烈的对比，大声传递出它的目的。例如，我在图 3-28 中使用了一种更大、更细、字母间隔更大的大写字体，让简历的标题对比更强烈。



Yevgeniy
BRIKMAN
ybrikman.com

SUMMARY
Programmer, writer, speaker, traveler.
I love to build beautiful software and products.

EXPERIENCE
Founder, *Atomic Squirrel*
2015 – Now
Atomic Squirrel specializes in helping startups get off the ground.

Author, *O'Reilly Media*
2014 – 2015
Wrote the book *Hello, Startup*.

Staff Software Engineer, *LinkedIn*
2009 – 2014
Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer, *TripAdvisor*
2008 – 2009
Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer, *Cisco Systems*
2006 – 2008
Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

EDUCATION
Master of Engineering in CS, *Cornell University*
2005 – 2006

Bachelor of Science in CS, *Cornell University*
2002 – 2005

图 3-28：标题字体对比更强烈的简历设计方案

对比还有另一个关键的作用——可以促使用户把关注点放在设计的某一个重要部分。虽然人们在读书时可能会阅读每一个字，但多数产品的用户却未必如此。

> 大多数时候，（用户）实际上（如果我们运气好的话）在每个新页面上只会瞥一眼，扫视一下文本，点一下令他们感兴趣或差不多像他们正在寻找的东西的第一个链接。页面上往往有一大片区域对用户来说是看不到的。我们想的是“伟大的文学作品”（至少也是“产品宣传册”），而用户的实际感受更像是“百公里时速一闪而过的广告牌”。
>
> ——Steve Krug，《点石成金》

因此，不仅设计中的每种字体都要服务于一个特定的目的，而且每一个界面都要有一个吸引用户操作的中心，这被称为**行为召唤**（call to action，CTA）。例如，我希望人们在 hello-startup.net 上做的主要事情是了解这本书，所以我加了一个大的“learn more”按钮作为一种 CTA，如图 3-29 所示。

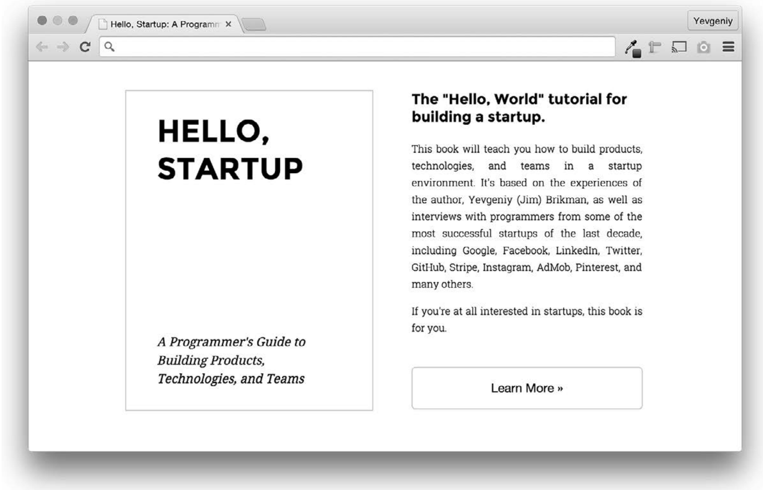


图 3-29：以 Learn More 按钮作为 CTA 的 hello-startup.net 设计方案

这仅仅只是第一步，我还可以使用颜色来增强对比，让这个按钮更加显眼。

6. 颜色

OKCupid 是一个通过颜色和对比很好实现 CTA 的出色示例，如图 3-30 所示。

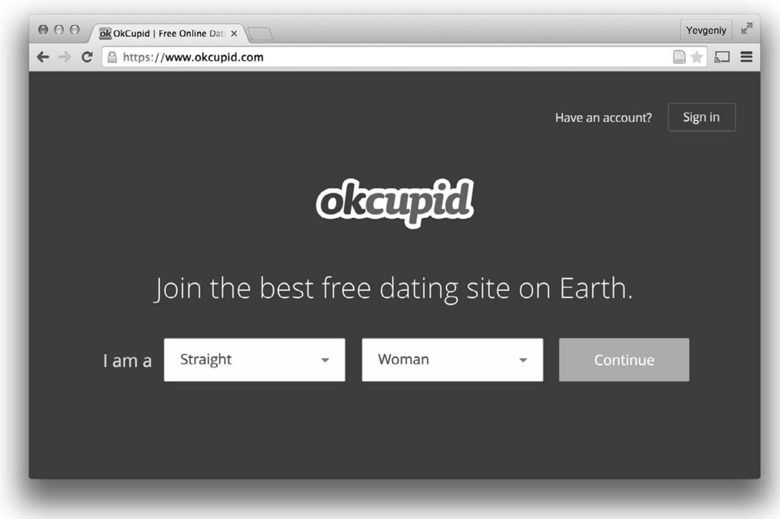
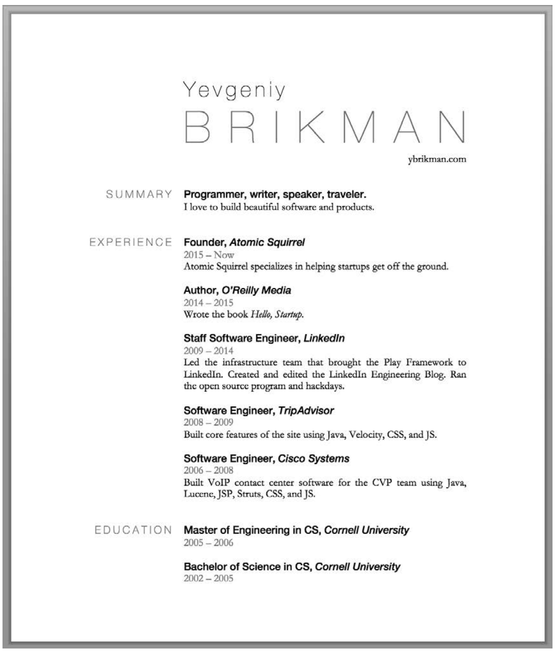


图 3-30：OKCupid 网站的 CTA

只要一打开这个网站，就很清楚网站是干什么用的（归功于清晰的文字说明），也知道应该做什么（归功于清晰的 CTA）。正中间的位置、大字体以及高对比度的颜色使得 CTA 一下子就映入眼帘。这就是有效使用对比的关键所在：如果页面上的两种元素是不同的，就要让它们具有非常大的差别。或者正如 William Zinsser 所写的："别稍微加强，直接加强。"

我们又怎么知道用哪些颜色可以达到好的对比效果呢？如果是小孩，会觉得用颜料和蜡笔涂色是很好玩的；如果是大人，在设计作品中挑选可用的颜色就没那么好玩了。颜色理论甚至比排版还要复杂，想要做好，必须考虑一些生理学因素（例如把红色文字放在蓝色背景上可以产生一种称为“色彩实体视觉”的效果）会让文本变得模糊，使阅读困难甚至痛苦；还要考虑一些生物学知识（例如有 8% 左右的男性有色弱问题，但有 2%~3% 的女性有额外的颜色感知能力，能够比一般人看到更多的颜色）、心理学的知识（例如每种颜色和情绪都有一定的关联，并且对情绪会有某种特定的影响）、技术问题（例如在数码显示中要使用 RBG 色彩模型，而多数的打印设备使用的都是 CYMK 模型）、艺术感受力（例如有些颜色能够和谐地放在一起，其他颜色则未必），以及色彩的物理原理和构成机制（例如色轮、原色、二级色和三级色、混色、色相、饱和度、亮度、色彩和色度）——有许许多多需要我们学习的东西。

如果你才接触颜色，我可以告诉你两个节省时间的小窍门。第一个窍门就是先用黑白图进行设计，然后再添加颜色。也就是说，先准备好文字材料、布局和排版，不给设计作品添加颜色。等到最后，其他所有东西都已经准备就绪，就可以添加颜色了，那时就只有这一个目标。我们可以把上色想象成给房子上漆：应该把墙、窗户和门都建好以后再给房子上漆，而不是之前就刷好。如果设计的其余部分已经完成了，我们可以更容易地试验不同的色彩方案，可以有意地选择一些颜色去烘托或营造一种氛围，或者表现某一特定的主题。例如，我们用来演示的简历一直都是黑白的，现在可以很轻易地给它加上一种颜色，起到突出的效果，如图 3-31 所示[5]。



Yevgeniy
BRIKMAN
ybrikman.com

SUMMARY
Programmer, writer, speaker, traveler.
I love to build beautiful software and products.

EXPERIENCE
Founder, *Atomic Squirrel*
2015 – Now
Atomic Squirrel specializes in helping startups get off the ground.

Author, *O'Reilly Media*
2014 – 2015
Wrote the book *Hello, Startup*.

Staff Software Engineer, *LinkedIn*
2009 – 2014
Led the infrastructure team that brought the Play Framework to LinkedIn. Created and edited the LinkedIn Engineering Blog. Ran the open source program and hackdays.

Software Engineer, *TripAdvisor*
2008 – 2009
Built core features of the site using Java, Velocity, CSS, and JS.

Software Engineer, *Cisco Systems*
2006 – 2008
Built VoIP contact center software for the CVP team using Java, Lucene, JSP, Struts, CSS, and JS.

EDUCATION
Master of Engineering in CS, *Cornell University*
2005 – 2006

Bachelor of Science in CS, *Cornell University*
2002 – 2005

图 3-31：加上强调色彩的简历

注 5：请读者登录图灵社区本书页面，在随书下载栏目免费下载查看该彩色图片：www.ituring.com.cn/book/1776。
——编者注

需要注意的是，只有在简历的布局（两栏）和字体（大字号、字母间距较大、字体较细的 Helvetica Neue 字体）都已经设置完成之后，颜色方案的选择才有意义。如果我尝试在原始设计中添加颜色，结果很可能是不一样的，而且等到布局和排版完成，肯定还得再做修改。

对于 hello-startup.net，我在整个设计中应用了灰度效果，让设计中的图片去决定设计所采用的颜色。例如，我为本书选择的封面图片有灰色倒影和绿色文本，所以在整个设计中都使用了这两种颜色，如图 3-32 所示[6]。



图 3-32：封面图片中的灰色和绿色决定了设计的其他地方所使用的颜色

第二个技巧就是使用专业人士提供的调色板，而不是靠自己去想。当然，你也可以模仿喜爱的网站的配色方案，但也有许多专用的工具可以帮助你运用颜色。例如，Adobe Color CC 和 Paletton 能够利用颜色理论为你提供配色方案（单色系、邻近色、颜色三角）。Adobe Color CC、COLOURlovers 和 Dribbble 提供的颜色搜索功能还可以对预设的一些颜色方案进行浏览。

3.1.4　视觉设计快速回顾

图 3-33 和图 3-34 分别展示了一份简历和 hello-startup.net 的设计过程。请读者花点时间看看这些设计图，有意识地指出它们之间的差异。

注 6：请读者登录图灵社区本书页面，在随书下载栏目免费下载查看该彩色图片：www.ituring.com.cn/book/1776。——编者注



图 3-33：简历的设计过程

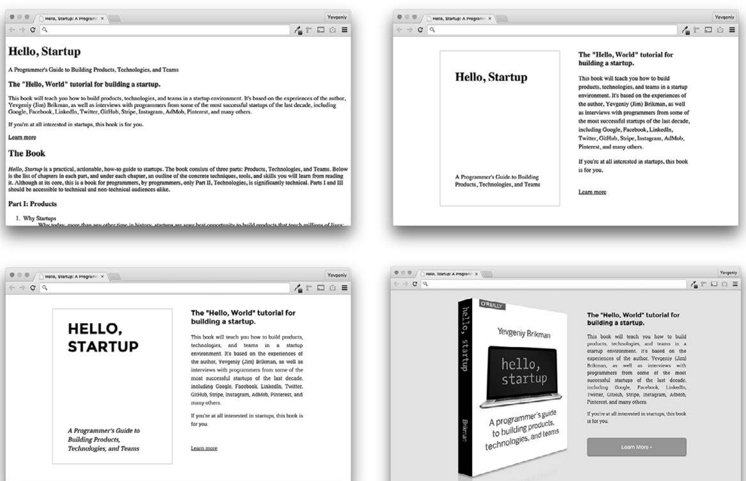


图 3-34：hello-startup.net 的设计过程

我希望大家能够从上图中找出视觉设计的以下几个因素：

- 左上：文案；
- 右上：布局（对齐和亲密性）；
- 左下：排版（行宽、行距、字型、字体）；
- 右下：对比与色彩。

最后我想说，所有设计步骤都利用了我在网上找的一些模板、字体组合和调色板。所以，设计重用还是这一切的核心。

3.2 MVP

在创业时面临的第一个设计上的挑战，就是要实现产品最初的版本。即便你已经想出了出色的点子，也对真实的客户进行了验证，你也要耐住性子，把自己锁在房间里，花上一年时间去设计，才可能做出完美的产品。但是请记住，产品并不仅仅是一个点子，而是新的问题、新的想法和执行的不断循环。执行是昂贵的，所以你需要尽可能低成本、快速地向客户验证你遇到的每一个新问题和想法。最好的方法就是实现所谓的**最简可行产品**（minimum viable product），或者叫 MVP。

MVP 是一个经常会被误解的术语。“最简”（minimum）通常会被误读为“要尽你所能快速发布任何东西”；“可行”（viable）通常会被误解为“足够让产品起作用的功能”，这样会误导人们实现很多没有必要的功能，却忽略了实际上要紧的东西；而“产品”（product）则错误地暗示了 MVP 必须是一个产品，所以人们经常忽略对一些更简单、成本更低的点子实现 MVP。

MVP 这个术语是通过 Eric Ries 的《精益创业》一书普及开来的，他在书中给出了一个恰当的定义：MVP 是“新产品的某一个版本，团队可以利用它以最小的付出去最大程度上、验证性地了解客户”。MVP 的关键就是从中**学习**，其目的就是找到成本最低的方式去验证对真实客户的假设。

MVP 中的“最简”意味着所有对当前假设的验证没有直接帮助的东西都应该被排除。例如，当 37signals 最早推出他们的项目管理工具 Basecamp 时，他们的 MVP 并不具备要求客户支付的功能。他们所要验证的假设是客户会注册使用一个基于网页的、具有简单用户界面的项目管理工具。而支付系统对于这样的验证并没有帮助，所以只能从 MVP 中排除出去，后续再添加（如果客户在实际中开始注册的话）。换句话说，他们把时间花在找出既整洁又简单的 MVP 设计上，因为那是他们所要测试的基本假设。

MVP 中的“可行”意味着 MVP 能够让客户接受它。MVP 也许有 bug，也许还缺少某些功能，也可能外观不怎么好看，甚至和最终实现的产品根本就不一样，但是它解决了客户所关心的问题。图 3-35 很好地展示了不可行的 MVP 和可行的 MVP 之间的差别。

最后，MVP 中的“产品”实际上是“试验”。它可以是产品的工作原型或者更简单的东西，比如带有演示视频的登录页面——只要它可以验证你的假设就可以了（阅读 3.2.1 节了解更多信息）。

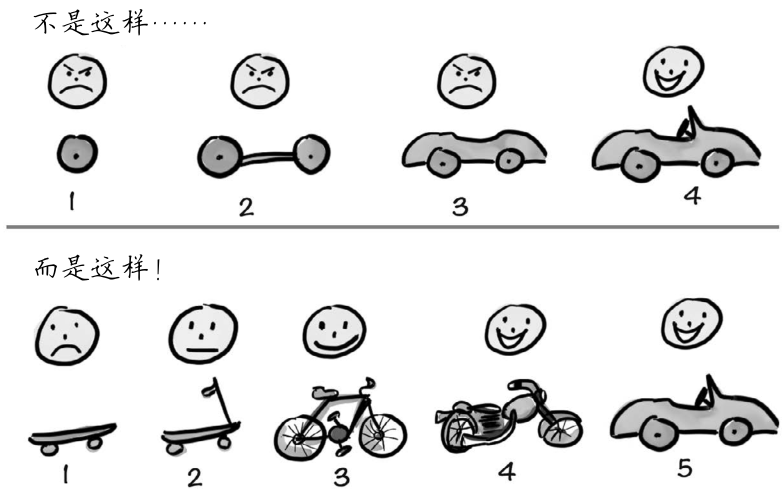


图 3-35：如何实现可行的 MVP，图片由 Henrik Kniberg 提供

MVP 的实现并不是一次性的行为。对某种东西而言，在找到可行的产品方案之前，极有可能需要实现多个 MVP。但更为重要的是，MVP 的构建不仅仅是在产品生命周期的早期所要做的一件事，它更多的是一种思考方式。不妨把它想成在玩纸牌游戏，每次都下一小点赌注，而不是一次就把房子全压上。不论你是在试验从未有人用过的新产品的点子，还是是为已有大量用户的产品添加新的功能，都应该有使用 MVP 的习惯，我们可以把 MVP 的实现归纳为以下几点：

(1) 找出风险最大、最重要的设想；
(2) 把这种设想以一种可测试的假设描述出来；
(3) 构建一个最小的实验（一个 MVP）去测试你的假设；
(4) 分析结果；
(5) 用新发现去重复第一个步骤。

不管对一个点子有多么自信，一定要努力找到最小、成本最低的测试方法，而且要随时保持项目规模小、可改进。Standish 集团通过对 50 000 多个 IT 项目进行研究，发现有 3/4 的小项目（少于 100 万美元）可以成功地完成，只有 1/10 的大项目（大于 1000 万美元）能够按时且在预算内完成，而超过 1/3 的大项目是彻底失败的。

> 通过深入研究，Standish 集团明确指出，项目成功的秘密就是要坚决设立并实施对其规模和复杂度的限制。这两点是成功的最关键因素。
>
> ——The Chaos Manifesto 2013

接下来讲讲可以构建哪些类型的 MVP。

3.2.1 MVP的类型

MVP 并不一定是实际的产品，它只需要能够在客户使用的时候验证你的假设就可以了。以下是最常见的 MVP 类型。

展示页面

有一种比较容易实现、成本低、效果又出众的 MVP，那就是做个简单网页，描述产品情况并在用户感兴趣的时候让他们提交某些信息，比如让用户提供 email 地址以便获得更多信息，或者让用户进行预订。总的思路就是向用户描述产品最理想的景象，看看它对用户有多大吸引力，哪怕产品尚不存在。如果你以最理想化的方式向用户描述了你的点子都无法说服一小部分人在你的邮件列表上注册，也许就需要再重新想想。例如，社交媒体管理应用 Buffer 开始时就用一个页面展示了有关产品的理念和价格细节，并提供了注册邮件获取更多信息的功能，如图 3-36 所示。他们获得了足够多的注册量，更重要的是，他们在价格选项上也获得了足够点击，令他们足以信心满满地去实现真正的产品。

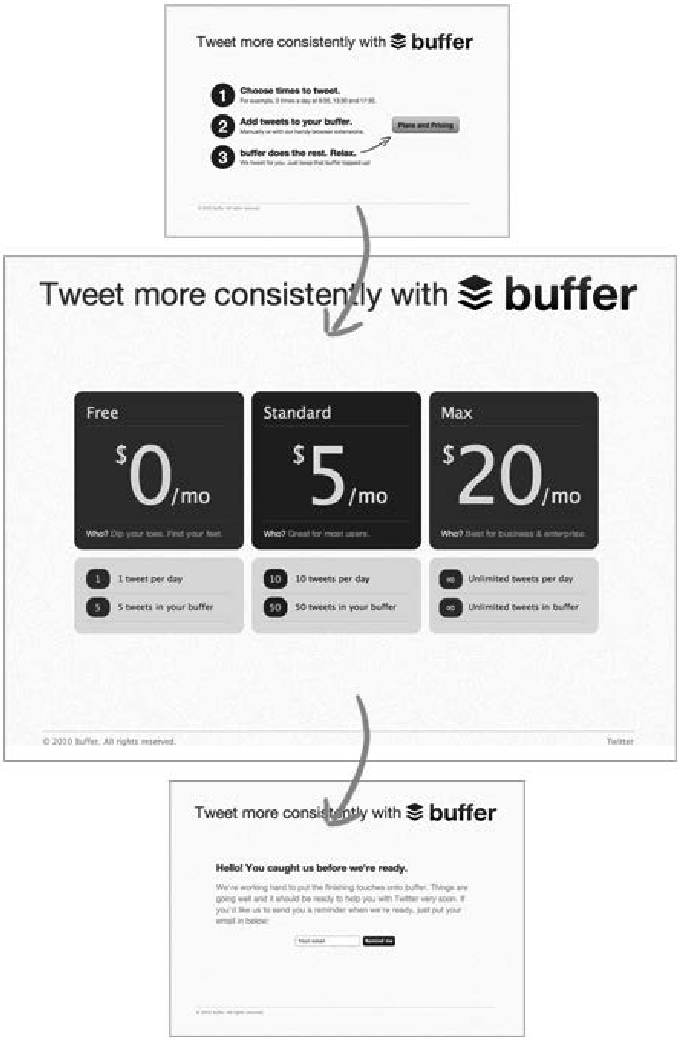


图 3-36：Buffer 的 MVP

与可用产品相比，展示页面上的文本和图片的更新速度更快，所以这也是探索设计、广告词和市场的最有效方式之一。我们可以对不同的表达方式进行试验，可以尝试针对不同的消费群体做工作，也可以尝试不同的价格策略，不断地重复，直到找到最有效的结果（阅读第 4 章了解如何评估每一次试验的表现情况）。我们也可以把自己的展示页面放在 AWS 或 GitHub Pages 上，或者使用各种专为实现展示页面而定制的工具，比如 LaunchRock、Optimizely、Lander 或者 LeadPages。

介绍视频

在 Drew Houston 开始构建 DropBox 之前，他想确认自己不会花多年心血却做出无人问津的产品。但即便实现一个用户可以在自己电脑上试用的简单原型，也要花很长的时间，因为想要存储所有的数据，就需要搭建起一个可靠的、高性能的在线服务系统。Houston 选择的替代方法是建构一个简单得多的 MVP：一个具有注册表单，还带有 4 分钟讲解视频的展示页面，如图 3-37 所示。



图 3-37：DropBox 介绍视频

这段视频是介绍产品实际用途的一种很有效的方式，它不仅能描述产品的功能，还有一些为懂技术的观看者制作的复活节彩蛋（例如引用了《XKCD》漫画和《上班一条虫》中的内容）。Houston 把视频放在 Hacker News 和 Digg 网站上，在短短 24 小时内，展示页面就有数十万的访问量和 7 万人注册。这给了 Houston 很大的信心——实现真正的产品是值得的。还有像 PowToon、GoAnimate 和 Gamtasia 这样的工具，可以免费或以很少的预算制作出介绍视频。

众筹

Kickstarter 或 Indiegogo 这样的众筹网站有点像带有介绍视频的展示页面，只不过对项目感兴趣的客户会给你钱支持你的项目，而不是只填写 email 地址。换句话说，这是一种让客户在你实现产品之前就掏钱购买产品的方式，也是最好的一种验证方法。最成功的 Kickstarter 众筹活动之一就是 Pebble 手表，该项目差不多只用一个原型就从 68 000 名赞助人手中筹集到了 1000 万美元，如图 3-38 所示。

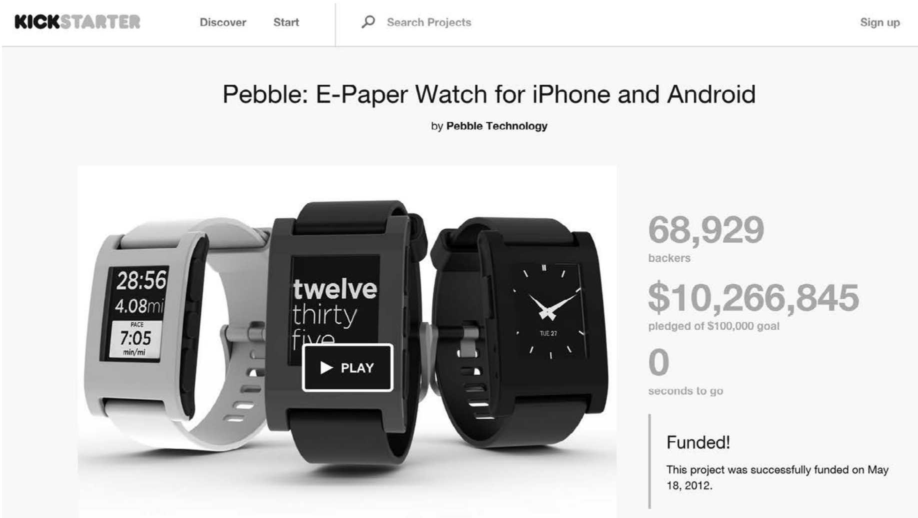


图 3-38：Pebble 在实现产品之前就在 Kickstarter 上筹集到了 1000 万美元

绿野仙踪

所谓 MVP 的绿野仙踪童话，就是呈现给用户一个看似真实的产品，但幕后却由创始人手动实现所有的一切。例如，在 Nick Swinmurn 测试 Zappos 的点子（在网上卖鞋）时，他到当地的鞋店，将他们现有的鞋都拍了照片，然后把照片挂到网站上，让网站看起来就像一个真正的在线鞋店，如图 3-39 所示。

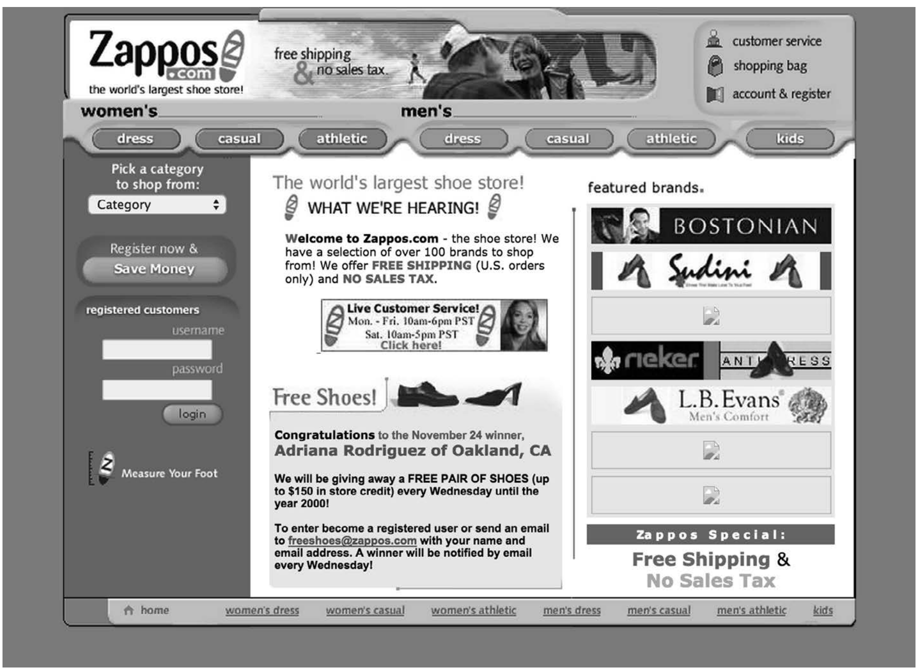


图 3-39：利用 Internet Archive 找到的 1999 年的网站截图

当用户在网上下订单的时候，Swinmurn 到当地的鞋店把鞋子买下，发给客户。Swinmurn 验证“人们乐意在互联网上买鞋”设想的这种方法既不需要大量存货，也不需要拥有自动订单系统以及储存、配送鞋子的工厂。人们并不会关注你在幕后是怎么运作的。

拼凑式 MVP

拼凑式的 MVP 类似于绿野仙踪式的 MVP，只是拼凑式的 MVP 会尽可能以低成本的方式、用已有的事物自动实现人工实现的那部分。例如，为了制作 Groupon 的 MVP，Andrew Mason 把定制的皮肤加在一个 WordPress Blog（见图 3-40）上，用 File Maker 去生成优惠券的 PDF，再用 Apple Mail 发送出去。



图 3-40：利用 Internet Archive 找到的 2009 年的 groupon 网页截图

读者可以访问 http://www.hello-startup.net/resources/mvp，上面提供了构建 MVP 的一些工具列表。无论最终要实现哪种类型的 MVP，关键是确保自己实现的是简化但仍然可用的东西，而最好的方法就是关注自己产品的差异性。

3.2.2 关注差异性

在一个名为“You and Your Research”（你和你的研究）的演讲中（该演讲经常被俗称为“你和你的职业”，因为它的建议不仅可以用到研究中，也可以用在几乎所有职业上），贝尔实验室的著名数学家 Richard Hamming 讲述了他在午餐时和一些化学研究员坐在一起发生的事。

> 一开始，我问道："在你们的领域中，什么是最重要的问题？"差不多一周之后，我又问："你们正在研究的最重要的问题是什么？"又过了更长的时间，有一天我过去又说："如果你们正在做的事情不重要，而且你们觉得它也不会变得重要，为什么你们要在贝尔实验室里研究它？"从此以后，我就成为不受欢迎的人了，吃饭的时候只好找其他人坐在一起。
>
> ——Richard Hamming，You and Your Research

Richard Hamming 在贝尔实验室完成了一些重要的工作，因为他专门找出一些重要的问题，而不是挑感觉舒服的问题。尽管他提问的方式可能会让人不舒服，但这种方式是我们应该在生活中应用的。什么是自己所在领域中最重要的事情？自己正在做的事情又是什么？为什么它们会不一致呢？

同样的道理也可以用在制作 MVP 上。对你的产品而言，什么是重要的问题？你在 MVP 中实际实现的又是什么？为什么会不一致？产品最重要的一点就是**差异性**：让产品和其他替代品区分开来的特性。人们通常把差异性称为"竞争优势"，但这个词听起来就像其他优势一样，不管超过多少，只要具备就足够了。事实并非如此，你的差异性必须比竞争者多得多才行。你要寻求的不是 10% 的改进，而是 10 倍的改进。做不到这点，大多数客户就觉得不值得花工夫换成你的产品。

因此，很重要的一点就是要问自己：我的产品有哪两三个地方是做得特别出色的？只要你找出了这样几个核心特性，就可以以此做出你的 MVP，先忽略其他东西。例如，当 Google 开始推出 Gmail 的时候，它的差异性就是提供 1GB 的存储空间（那个时期其他大多数 email 提供商只会给你 4MB）以及灵活的用户界面（拥有对话视图、强大的搜索功能，使用 Ajax 技术即时显示最新的 email，不需要非得刷新页面才能看得到）。而其他几乎所有的特性，比如"富文本"的编辑器和地址本，都是最简单的实现或者就没有——但这些都无关紧要，因为它提供的差异性如此引入注目，使得其他所有的 email 服务看起来都变得黯淡无光。

另一个很好的例子就是最早的 iPhone。Apple 以提供完整、优美、端到端的解决方案而著称，但从很多方面来看，最早的 iPhone 就是一个 MVP。它没有应用商店、GPS、3G、前摄像头，没有后摄像头闪光灯、游戏、即时消息、复制和粘贴、多任务、无线同步、Exchange 邮件、彩信、蓝牙立体声、语音拨号、音频录制或视频录制。尽管这样，iPhone 仍然领先其他智能手机好几年，因为 Apple 持续不断地把注意力放在如何把少数几件事情做得异常出色上：它在多指触碰用户界面、硬件设计、音乐及上网体验上至少比其他手机好上 10 倍，让顾客爱上了它。

让客户爱上你的产品，而不只是喜欢它，这是一个巨大的优势。让一个已经有少量用户爱上的产品变得有更多的用户爱上，比起让大量的用户从喜欢一个产品变成爱上一个产品，前者要容易得多。让一个用户从"喜欢"到"爱"，你要让他们大为心动才行。你需要让他们能大叫一声由衷地赞叹，想想最后一次有东西让你发出赞叹的感觉，很可能是有人超出你的预期，让你高兴不已，也可能是一些超乎寻常的东西。因为做出不同一般的东西本身就要花大量的时间，所以，如果你想让用户能够爱上你，比起让许多事情都差强人意，你应该让少数事情无与伦比。

那么，我们怎么知道要把关注点放在哪些特性上呢？有一种方法，就是在做出产品之前，

先写一篇宣布产品发布的博客，看看文章中有哪两到三个关键特性是你会重点宣传的，你会在插图中展示哪些特性，博客的标题会是什么。好的博客文章都是简短的，所以这样的训练可以帮助我们梳理出哪些特性真正能让产品充满诱惑力。这样的特性就是 MVP 必不可少的，其余的一切都是可选的。事实上，其余的一切不仅仅是可选的，大多数时候，甚至对产品是有害的。每一个额外的特性都会带来显著的成本（阅读 3.1.2 节了解更多信息），所以，除非该特性对取悦客户或者验证假设是绝对不可或缺的，否则就不应该放到 MVP 中。

只要找出产品的差异性，并且以此制作 MVP，我们就可以利用它去验证假设。而最为重要的验证，就是让顾客去购买我们的 MVP。

3.2.3 购买MVP

我们要对 MVP 确定一个目标，即便在很早的阶段，也要让客户购买你的解决方案。注意，这里强调了“购买”一词。许多人会告诉你他们“喜欢”一个点子，甚至也许想得到它。但是，喜欢某种东西和承诺会购买某种东西是大不相同的。购买一种新产品不仅仅要花费金钱，还要花费时间——他需要花时间去说服家人（如果是消费产品的话）或者同事（如果是企业产品的话），让他们相信产品是值得的；而且，还要花时间去安装和部署，花时间培训自己和别人去使用它，将来还得花时间去维护和更新。即便你的产品对某些用户是免费的（例如靠广告支持的网站或者免费增值服务），他还是要付出自己的时间，而时间因素也会让他们考虑一番。所以，不管你考虑采用什么样的定价策略，目标就是要让客户牢牢承诺会购买你的产品。

之前介绍的每一种 MVP，即便是最简化的类型，也为客户提供了购买的机会。显然，这就是众筹 MVP 的意义所在。但是，我们还可以在展示页面类的 MVP 上提供预订表单，也可以用绿野仙踪式的 MVP 去收费，哪怕不得不接受现金支付。我们也可以不断调整价格，直到找到最佳价格，但别把它免费提供给别人。事实上，调整价格就是一种很好的方法，可以了解客户对待产品的认真程度。

> 我问（我的客户）“如果产品是免费的，有多少人会真的购买或使用”的目的就是把价格因素抛开，看看产品本身是否能够让客户心动。如果做到了，我会接着问几个问题：“好了，这个产品不是免费的。事实上，假设我要收取你们 100 万美元，你们还会购买吗？”虽然这种对话听起来有点儿像开玩笑，但我一直在用这种方法。为什么呢？因为超过一半的时候，客户会像这样说：“Steve，你怕是疯了吧。这个产品不会值 25 万美元以上。”其实，我只不过想让客户告诉我，你们愿意支付多少钱而已。
>
> ——Steve Blank，《四步创业法》

什么样的客户会承诺购买并不存在的产品呢？或者这么问，即便是可用的原型，什么样的客户乐意用你全新的创业产品去开展业务，而不顾各种 bug、性能问题、缺失的功能，而且你还有可能在几个月之后就歇业了？在《创新的扩散》一书中，Everett Rogers 把客户分成了 5 种类型。

(1) **创新者**愿意承担新技术的风险，因为技术本身就是他们生活中的主要兴趣，不管功能如何，他们总是留心寻找最新的发明。

(2) **早期采用者**也愿意承担新技术的风险，不仅仅因为他们对技术感兴趣，还因为他们很容易联想到该技术将会给生活带来什么样的好处。
(3) **早期的大多数**客户是迫切需要解决具体问题的。他们能够想象到新技术是怎样成为解决方案的，但又知道许多新的技术革新最终都会失败，所以在自己购买新技术之前，他们更愿意等待，看看该技术是否能解决他人的问题。
(4) **后期的大多数**客户也有需要解决的具体问题，但他们不喜欢使用新技术去解决问题。他们更愿意等到一项技术成熟，自身已经成为标准，并且已经具备了很好的支持体系才会购买。
(5) **滞后者**会尽可能避免使用新技术。他们是最后采纳新发明的人，而且通常都是在别无选择的情况下才会采纳。

每种类型客户的数量在大体上遵循钟形曲线分布，如图 3-41 所示。

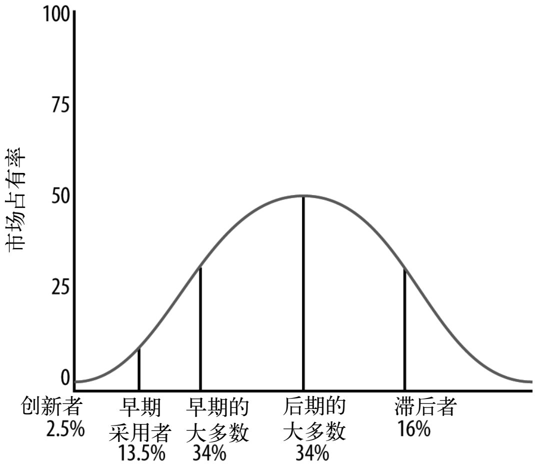


图 3-41：创新的扩散

要想成就一家成功的公司，通常来说，我们必须把产品销售给早期和后期的大多数客户。但只有说服创新者和早期采用者购买我们的产品，公司才能达到这一阶段。也就是说，创新是沿着图 3-41 中的钟形曲线从左到右扩散的，只有在前一阶段取得成功，才能够跳入新的阶段中。而且每种类型客户的要求是不同的，所以，理解目标客户的类型就至关重要了。否则，将产生错误的产品、错误的营销策略和错误的销售方法。

> 早期采用者的购买……就是很有效的**变革推动**。早期采用者希望成为他们行业中首先实现变革的，不论是降低产品的成本，还是快速占领市场、提供完善的客户服务，或者形成其他一些可比较的商业优势，凭此在竞争中取得领先地位。他们希望在旧方法和新方法之间有彻底的分割，他们已经准备好与根深蒂固的抵抗力量做斗争并捍卫这一过程。作为第一个吃螃蟹的人，他们也做好了心理准备，去忍受必然出现的 bug 以及任何刚投向市场的革新所带来的各种小毛病。
>
> ——Geoffrey Moore、Regis Mckenna，《跨越鸿沟》

在创业早期仍在验证问题和解决方案的时候，我们的目标就是找到合适的早期采用者。他们是在我们的解决方案远未准备好之前就会承诺购买的客户，因为他们相信的是我们

提出的愿景而不是具体的产品。Steve Blank 把这样的客户称为**早期传教士**，并提供了一份实用指南，帮助我们找出这样的人。

- 他们有问题或需要。
- 他们明白自己遇到了问题。
- 他们正在积极地寻找解决方案，并且设定了找到方案的时间表。
- 问题让他们非常痛苦，所以他们已经拼凑出了临时的解决办法。
- 他们已经承诺购买，或者可以在短时间内获得或申请到预算去购买。

——Steve Blank，《四步创业法》

如果你发现了一个已经自己鼓捣出临时解决方案的客户，这也许就是最好的信号了，因为你已经发现了真正的问题，已经提前迈出了一步。每个行业都有早期传教士，尽管这样的人通常都不会很多。也就是说，在早期还没有产品和客户的时候，即便能够得到一个客户都是个巨大的胜利。在我们获得数千或数百万的客户之前，要先得到一个客户，才能一而十、十而百……为此，我们可能不得不做一些无法规模化的事情。

3.2.4　创业须从无法规模化的事情做起

Y Combinator 最经常给创业公司提供的一条建议就是：创业须从无法规模化的事情做起。也就是说，在创业的早期，我们也许不得不亲自动手做很多事情，比如招聘人员、招募用户以及提供客户服务。这些事情会让人感觉效率低下，特别是对程序员来说，他们总是会叫嚷："但这些都是无法规模化的事情！"可是，这些体力劳动通常是让公司这一机器转动的唯一途径。而且只有在公司发展起来之后，我们才能去考虑规模化的事情。例如，Airbnb 的创始人在纽约挨家挨户地招募早期用户，甚至帮他们给公寓照相；Homejoy（一家帮助客户寻找家居清洁服务的公司）的创始人最初会去到客户的家里，自己进行全部清洁工作；Pinterest 的创始人会去咖啡店里亲自让陌生人使用他们的产品，也会去 Apple 的体验商店把所有的浏览器首页都设置为 Pinterest；Wufoo（一家帮助用户创建在线表单的创业公司）的员工过去经常会把手写的感谢信送到每一个客户手中。

> 也许影响创始人意识到能够在多大程度上关心用户的最大障碍，就是他们自己从来没有体验过这样的关心。他们对待客户服务的标准是依据他们自己作为客户的那些公司的标准来设定的，而那些通常都是些大公司。蒂姆·库克不会在你买了笔记本电脑之后给你寄一张手写的卡片——他做不到，但你可以。这就是小公司的好处：你可以提供大公司实现不了的服务。
>
> 一旦意识到现有的一些习惯做法并没有超出用户的预期体验，不妨好好想想，我们可以用多少手段去取悦用户，这是很有意思的。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

当你还是一家小型创业公司，仍然还在验证自己的点子时，做一些无法规模化的事情去获得早期的客户是你可以承担的方式。如果点子可行，后续可以通过自动化的方式，让这一过程变得更具扩展性；但如果点子不可行（大多数点子都是这样的结果），那么你也节省了大量的时间，因为你不需要为了错误的事情而做一大堆自动化工作。另外，这种方式可以让你直接接触业务的烦琐细节，你会成为领域的专家，而这一点在前面已经说过，它对于想出伟大的点子是至关重要的。

3.3 小结

> 用户界面就像讲笑话，如果非得解释清楚，就不那么好玩了。
>
> ——Martin Leblanc，iconfinder 创始人

设计是一项重要的技能，因为用户界面就是产品。好在设计过程是迭代的：任何设计都可以递增式地改进，任何人也都可以递增式地提高自己的设计技能。而最好的做法就是去重用现有的设计、编写用户故事并为人物角色而设计。通过实践，我们学到了文案、布局、排版、对比、重复和颜色的知识。我们为产品赋予了个性，特别是认识到有礼貌的产品应该是积极响应、考虑周到和宽容大量的，我们设计出能够在情感上和用户产生共鸣的产品。通过不断进行可用性测试，我们可以获得如何取得进步的直接反馈。

但不管设计做得多么出色，我们还是无法确定产品能否成功。因此，最佳的策略就是不断进行一些小的试验，根据真实用户的反馈进行调整。进行小规模的市场研究，与潜在的客户进行交谈，发布快捷的 MVP，从用户的反应中学习，然后不断地重复，周而复始。

> 我是 Amazon 上销量最好的面试技巧类图书的作者，但这一切都是从一份仅有 20 页的 PDF 文档开始的。老实说，那份文档并不是很好，我现在看到都会感到尴尬。但作为一个 MVP，它已经足够了，即便那时候我并没有这样的想法。我用它对市场进行了测试，确定了这是一个真正的需求，可以在它的基础上进行扩展。它也让我很早就获悉了“哪些内容才重要”的读者反馈。
>
> 我创办的另一家公司的情况也差不多。开始时规模很小，后来我偶然才意识到自己有一家公司了。
>
> 自己或别人想出点子时，我们很容易找出各种理由把它作为一个不好的点子划掉。我可以告诉你一百万条我的公司为什么本来应该失败的原因，但是纵有这些原因，它还是成功了（某些情况下，也许正是因为这些原因的存在）。
>
> 真实的情况就是很难去预言什么是可行的，什么是不可行的。而 MVP 可以让你相对迅速和低成本地去尝试，这些结果通常比你的其他预测更有意义。
>
> ——Gayle Laakmann McDowell，Careercup 创始人、CEO

许多人都会觉得这种忙乱、不断试验和发生错误的方法会把事情弄得乱七八糟。他们看到成功的产品，总会以为它一开始就是这样出现在创造者的大脑中，完全成型、漂亮并完整。这好比看到迈克尔·乔丹轻松地统治着篮球场，就设想他从妈妈肚子生出来时就是 6.6 英尺高、216 磅重，就能够扣篮并做出无法阻挡的后仰式跳投。

> 在我的职业生涯中，有 9000 多次投篮没有命中，在近 300 场比赛中失利。我背负着信任去投制胜球，但是有 26 次没有投中。我的生命中有一次又一次的失败，这就是我为什么成功的原因。
>
> ——迈克尔·乔丹

只要你手中拿着的是一件精雕细琢的产品，你就要记住，其实我们看到的是无数次试验和错误的迭代之后形成的结果。这其中包含了许多失误、原地打转、重新设计和妥协折

中。这一路上，制造它的公司可能要为生存而苦苦挣扎，希望能够在倒闭之前找对路子。

> 开创一家公司就像把自己丢下悬崖峭壁，还要在下落的过程中把飞机组装出来。
>
> ——Reid Hoffman，LinkedIn 联合创始人、主席

这就是创业公司总是处于“搜索模式”的含义所在。这是一场和时间的疯狂比赛，你要尽快找到值得解决的问题，找到值得实现的方案。而实现这一切的最佳方式并不是寄希望于尤里卡时刻，而是要利用迭代、试验的方法去实现。

第 4 章

数据与营销

我们在上一章学习了如何根据自己的创业点子设计出 MVP。在本章，我们将学习如何利用数据与营销去完善 MVP。

数据就是如何把一些假设和猜测转变为具体、可操作的行为。我会讲讲为什么测量总是比不测量更好，也会详细介绍在所有创业过程中都应该跟踪的度量指标，谈谈如何应用数据驱动开发的方法，利用这些指标更好地做出决定。

营销就是如何让用户找到你的产品。如果你创造了一款不可思议的产品，却没有人真正知道它的存在，那这一切就没有意义了。我会逐一介绍创业公司可以使用的绝大多数常见的营销策略，包括口碑营销、市场推广、促销和品牌化。

4.1　数据

> 产品经理的工作就是要把两件简单的事情说清楚：
>
> - 我们正在进行什么比赛？
> - 我们怎么得分？
>
> 把这两件事情做对，就可以不经意间聚集一批在技术、运维、质量、设计和市场推广上具备天赋的杰出人才，在同一个方向上聚力前行。没有这两点，无论做多少优化和执行管理，都拯救不了你。
>
> ——Adam Nash，Wealthfont 主席、CEO

如果想要构建成功的产品，就必须知道你在进行的是什么比赛，如何得分。对公司来说，“比赛”其实就是“使命”的另一种说法，9.2 节将做详细介绍。现在，我想先关注如何得分。尽管有时候我们唯一有意义的测量手段就是直觉——比如在可用性研究中观察用户的愉悦程度。但在多数情况下，更好的记分方法是收集、分析数据。

现代软件创业公司的强项之一就是可以非常方便地收集业务方方面面的数据。使用 Google Analytics、KISSmetrics 和 New Relic 这样的工具（阅读 8.5 节了解更多信息），你

可以追踪到用户来自哪里、他们是如何使用产品的、哪个功能会带来最大的收益、技术栈的哪些部分拥有最好的性能，等等。我们可以借助这些数据，充分考虑后再决定做什么产品、使用什么销售渠道、技术如何进化，而不是盲目地猜测。在做决策时利用好数据，主要玩的就是测量的游戏。

> 测量：基于一个或多个观察结果定量地降低不确定性。
>
> ——Douglas W. Hubbard，《数据化决策》

从它的定义中可以发现，测量并不是要消除不确定性，而仅仅是要降低不确定性。我们永远不可能消除不确定性，不论是商业还是生活，任何地方都不可能是完全确定的。没有一种测量是完美的，但我们不能仅仅因为测量是不精确的，或者还存在一些不确定，就认为它是毫无价值的。不完美的测量通常比没有测量或者按自己的意见（哪怕是专家的意见）行事要更好。密歇根大学的研究人员收集的数百个研究结果表明，基本的测量和定量的分析通常比人类专家有更出色的表现。

> - 在预测大学新生的 GPA 时，对高中排名和能力测试做个简单的线性模型，就能胜过经验丰富的管理人员。
> - 在预测罪犯再次犯罪的可能性时，基于犯罪记录和监禁记录的预测就能好于犯罪学家的推测。
> - 在预测医学院学生的学习成绩时，基于过去的学习成绩所建立的简单模型比对教授进行访谈得到的预测效果更出色。
> - 在二战时，对海军新兵在训练营中的表现情况进行预测的研究表明，根据高中档案建立的能力测试模型比专业的面试官的效果更好。即便为面试官提供了相同的数据，当他们的专业意见被忽略的时候，预测的结果才是最好的。
>
> ——Douglas W. Hubbard，《数据化决策》

人类，即便是专家，都会非常非常频繁地犯错。数据和测量就是我们把事情做好的最好的工具。如果你不是数据分析专家，也不用担心。在创业公司中，需要测量的大多数东西都不需要用到复杂的工具或方法。我们的目的并不是要在科学期刊上发表什么，而是要收集一些数据，提高我们做出好决定的概率。要实现这一目标，简单、不完美的方法对我们来说一般已经足够好了。

此外，正如本书经常提到的，测量也是一个迭代的过程。我们不一定从第一天开始就要建立起完美的跟踪和分析系统，也不一定为了获得测量的价值就去测量所有东西。事实上，我们从最初几次测量中得到的回报通常是最多的，随着越来越多地使用精心设计的方法，我们得到的回报反而越来越小。可以先从小的测量开始，哪怕先跟踪一个单独的指标（阅读 4.1.1 节了解更多信息），然后再逐步改进方法，测量更多的内容。

当然，并不是所有东西都可以测量，或者都应该被测量。对于每一种数据 X，我们要问自己两个问题。

(1) 如果我可以测量 X，它至少会影响一个具体的决定吗？
(2) 该决定的价值超过测量 X 的成本吗？

如果这两个问题都无法回答“是的”，那么就不值得去测量 X。话虽这么说，大部分人并不清楚能够以最小成本、付出最少努力去测量什么东西。《数据化决策》一书介绍了如何对各种各样的概念进行量化，包括一些看似模糊和不可测量的概念，比如产品质量、品

牌认知、安全以及风险。

> 一切东西都可以测量。不管以什么样的方式，如果一种东西能够被观察到，它本身就提供了某种类型的测量方法。不管这样的测量是多么“模糊”，只要它可以让你知晓更多的东西，它就是一种测量。而正是那些最可能被认为是不可测量的东西，几乎都可以用相对简单的测量方法去解决。
>
> ——Douglas W. Hubbard，《数据化决策》

接下来看看几乎所有的创业公司都需要跟踪的一些指标。

4.1.1　需要跟踪的指标

创业公司应当关注的数字对每家公司来说都是不一样的，但是有几种类型的指标是所有公司都需要跟踪的：

- 获取（acquisition）；
- 激活（activation）；
- 留存（retention）；
- 推荐（referral）；
- 收益（revenue）；
- 神奇数字（the magic number）。

前五个指标——获取、激活、留存、推荐和收益来自于 Dave McClure 所著的 *Startup Metrics for Pirates* 一书，这 5 个词的字母缩写“AARRR”也很好记。[1] 最后一个指标——神奇数字，是从前 5 个数字得来的。这个数字可以为我们提供一个很好的全局视角，了解创业公司的发展情况。

1. 获取

我们应该关注的第一个指标是获取，或者说是用户如何找到你的产品。4.2 节将介绍，如果没有人能找到你的产品，那无论它多么出色都没有意义。为了帮助人们找到它，我们可以使用搜索引擎、广告、博客、email、TV 和社交网络这样的营销渠道。因为用户的获取是在漏斗的顶端，是我们实现用户增长最先要面临的问题，所以通常也是最难突破的瓶颈。唯一可行的方法就是针对不同的用户获取渠道进行不同的试验，仔细跟踪哪些渠道是可行的，哪些又不可行。

2. 激活

在用户发现了产品之后，接下来要跟踪的就是激活这一指标，这是测量有多少用户被你的产品所吸引，进行了账号注册、邀请朋友、执行搜索或者支付等动作。如果你传递了错误的消息，或者你的设计没有清楚地让用户知道要做什么，又或者你的用户获取渠道带来了错误的受众，用户也许会反弹，即在看到产品之后立即离开，不执行任何操作。

通常情况下，随着我们对产品的改进以及更有针对性地获得用户，产品的激活率也会随之上升，反弹率会随之下降。这是可以通过深度的 A/B 测试（阅读 4.1.2 节了解更多信息）去提升的一个指标。此外，我们也一定要根据用户获取的渠道对激活数进行分解，了解

注 1：通过 Google Analytics，无论是移动应用还是网页，我们都可以对这些指标进行跟踪。该软件是免费的，使用也很简单，是初期分析时不错的选择。

是否有某些渠道能够产生较高的激活率。也许通过 Facebook 广告带来的用户有 80% 的反弹率，而通过 Google 搜索带来的用户只有 50% 的反弹率。这样的话，你就知道需要调整 Facebook 的广告定位，或者完全停止使用广告，加倍投入去提高搜索排名。

3. 留存

下一个阶段就是让激活的用户回来并再次使用你的产品。在某种程度上，这也是一种获取，但是用户的留存通常利用的是不同的渠道，所以应该分别进行跟踪。大多数用户都会被许多事情分散注意力，所以他们记不起要继续使用你的应用或者回到你的网站，除非你特地提醒他们。这就是为什么所有产品都要你订阅他们的邮件新闻；每一家公司都要维护一个博客，提供有用的提示和建议；所有移动应用都要给你发送通知；许多游戏还设定了一定的时效性，要求你不断回到游戏中，否则就会丢失进度。已经激活的用户甚至也需要多次看到产品，才能坚持使用并把它作为日常习惯。

我们还要跟踪在一个星期、一个月和一年之后有多少访问者会回来。就像部分用户获取渠道会有更好的表现一样，我们也要跟踪哪些用户留存渠道是最有效的。最后，请务必根据激活用户和获取用户的方式对用户留存数字进行分解。例如，如果你在做一个具有社交功能的产品，也许会发现被朋友邀请过来的用户比直接注册进来的用户留存度要更高一些。这就是为什么大多数社交应用程序都会努力让你邀请自己的朋友，并让你在初始激活流程中与用户进行联系。

4. 推荐

顺着邀请朋友这一话题，我们来介绍用户推荐这一指标。在某种意义上，这也是另一种形式的用户获取，但我们关注的是一个确定的渠道：在产品现有用户的帮助下获取新的用户。这是值得单独拿出来说的，因为世界上的每一种产品，无论使用的是什么营销渠道，在很大程度上都还是依赖于口碑的（阅读 4.2.1 节了解更多信息）。这就是为什么许多公司都会向用户提供向朋友推荐产品的奖励，比如你每推荐一个朋友去注册 DropBox，DropBox 就会提供 500MB 的免费空间。

用户推荐指标的重要性不仅仅在于它是我们获得用户的一个来源，也是衡量产品质量的一个指标。除非你真的喜欢一个产品，否则是不会把它推荐给朋友的，所以用户推荐数的增长通常是衡量产品是否改善的一个好方法。因此，了解用户从什么渠道被推荐过来是至关重要的，这也解释了为什么“你是怎么知道我们的”是注册页面上一个非常常见的问题。

5. 收益

我们也应该跟踪一下我们到底赚了多少钱，这些钱是通过什么渠道获得的，比如销售、订购、广告、业务拓展。你可能要用你的收益数字去计算**客户生命周期价值**（customer lifetime value，CLV），这是估量一个客户在与你产生关系的整个生命周期中，你可以从他身上赚到多少钱的方法（如果用 Google 搜索一下，可以找到一些计算 CLV 的简单公式）。为了让商业模式取得成功，CLV 必须大于获得用户的成本，所以我们要认真地跟踪这两个指标。

另外，也别忘记根据其他指标对我们的收益数值进行分解。例如，如果研究 Zynga 这样的公司制作的手机游戏，会发现其一半收益来自于 0.15% 的玩家，这些玩家被称为“鲸鱼玩家”。在这样的商业模式中，弄清楚是怎样的用户获取策略，激活、留存和推荐策略吸引到了更多的鲸鱼玩家，是这种公司获得成功的唯一方式。

6. 神奇数字

每一个公司都有一个“神奇数字”。这个指标就是，一旦用户突破了这个指标，他们就会遇到“惊喜”时刻，最终“粘”上产品。例如，对 Facebook 而言，表示新用户成为高度参与的用户先行指标就是“在他注册的 10 天内联系 7 个朋友”的神奇数字；对于 Twitter，一名新用户只要关注了 30 人之后就很可能成为一名活跃用户；在 Slack，只要一个团队交换了 2000 条消息，他们中的 93% 就会一直成为 Slack 的用户。找出你的神奇数字，就可以让你的团队关注一个清晰、具体、容易测量的目标，简化公司的决策制定。对于一个项目，我们可以看看它是否会显著影响我们的神奇数字。如果是的话，就进行；不是的话，先放一放再说。

Andrew Chen 在 Quora 上发表了一篇关于如何找出公司神奇数字的优秀教程，其中的第一步就是找出衡量公司成功的指标是什么。当成功指标增长的时候，你的业务也会取得成功；当指标下降的时候，你的业务会随之失败。这个指标对于每个公司而言有很大差别，但应该是相当明显的。比如 Facebook 和 Twitter 的大部分收益来自于广告，所以它们的成功指标和用户参与度结合得非常紧密（例如用户在过去 28 天的周期内会回到网站多少次）；Slack 是一个订购产品，所以它的成功可能和有百分之多少的用户会成为付费用户有紧密关系；Etsy 是一家电商公司，所以它的成功指标可能和网站的交易数有紧密关系。

一旦找到衡量成功的指标，第二个步骤就是判断用户的哪些行为与成功指标的增长是有关联的。抓取一部分有代表性的用户，把他们的全部数据（例如获取指标、激活指标等）放入一张巨大的表格中。如果幸运的话，把用户的活动指标和公司的成功指标放在一起绘制出图表之后，会发现它们之间有非常明显的相关性。例如，如果把 Twitter 用户的关注人数和他们连续登录的天数放在一起对比做图，可以得到图 4-1 所示的图表，临界点大概就在 y 轴的 30~40。有时的结果并不会很明显，这就需要进行回归分析，找到理想的相关性。

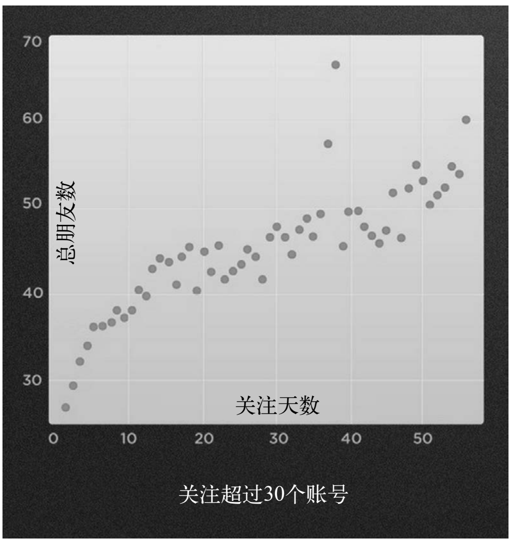


图 4-1：Twitter 用户关注人数与用户连续登录天数的对比

我们并不需要找到完美的相关性，也不需要把神奇数字定义为 25 个不同因素的集合——你的神奇数字只要能让你受到启发并且能帮你找到成功的原因就可以了。所以，我们更应该找出简单的指标。对于大部分变化，可以用这样一个能够解释多种变化的指标去解

释，而不是要弄得很复杂才解释得了。无论你选择什么指标作为神奇数字，最后一个步骤就是对它进行测试，确保它可以照预期影响你的成功指标。换句话说，我们要把原因和效果揭示出来，而不仅仅只是相关性。通常来说，我们可以通过 A/B 测试来得到这些数据，这是数据驱动开发的一个关键内容。

4.1.2 数据驱动开发

当你第一次开始收集业务指标的时候，其实就是一次大开眼界的经历。哪怕只是把 Google Analytics 挂到网站上，你也可以很清楚地了解有多少用户访问了网站，他们查看了什么页面，从哪里来，等等。关注数据一段时间之后，你会想要知道你所实现的产品和特性有哪些是成功的，如何通过数据提高成功的概率，而这就是**数据驱动开发**可以发挥作用的地方。

数据驱动开发有许许多多的内容，但我们最常用的就是 A/B **测试**。A/B 测试是一个营销学术语，表示一种受控制的实验，在实验中测试者被随机分成两组，即 A 组和 B 组，除了一个变量之外，两组之间其他所有变量都保持一致。这样就可以对这个独立的变量试验两个不同的值，一组一个，看看该变量对每一组行为在统计学上是否存在有意义的影响。当然，你可以对独立变量两个以上的值进行测试，但我们必须使用所谓的**对比测试**或**分桶测试**，那就不再是两个组，而是要把用户分成许多组（或者“桶”），然后测试独立变量的不同值对每一组的影响。

例如，大概在 2009 年的时候，LinkedIn 就在尝试订阅页面的新设计，让用户可以注册高级账户。这个设计要在右上角放一个快乐的人的大幅照片，但应该用什么样的人呢？我们可以让设计师根据直觉去挑选，但最终还是决定先进行分桶测试。我们准备了 4 种人的照片供选择，所以随机地把 LinkedIn 的会员分成了 A、B、C、D 四组，并分别向用户展示其中一张图片，如图 4-2 所示，你认为哪一组的表现最好呢？

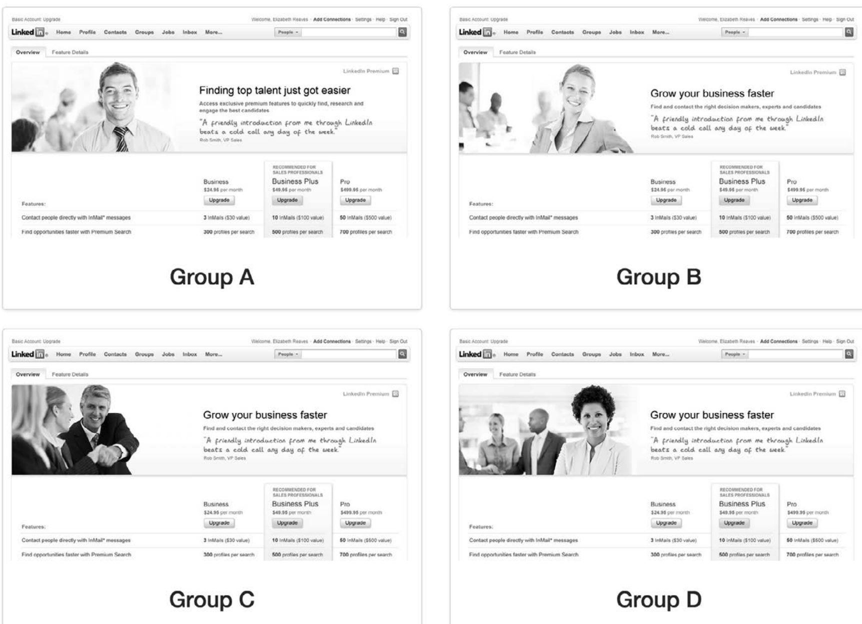


图 4-2：LinkedIn 订阅页面的分桶测试

结果表明，C 桶的效果明显胜过其他的分组。灰白头发男士的照片肯定让用户对我们建立了信心，因为当这幅照片出现在屏幕上时，订阅的用户远比其他分组多。不妨这样想：我们仅使用分桶测试来代替我们的直觉，就让我们赚到更多的钱。

一旦你尝到了 A/B 测试的甜头，就不愿再回去了。你会意识到，借助数据的力量，我们做出的决定将更加有效，你也会考虑把数据引入产品开发过程的方方面面。

1. 把数据引入产品开发过程

以下是 Etsy 开发产品的方式：

> (1) 实现功能；
> (2) 租用仓库准备发布派对；
> (3) 发布功能；
> (4) 举办发布派对；
> (5) 等待 20 个月；
> (6) 删除无用功能。
>
> ——Dan Mckinely，Etsy 和 Stripe 软件工程师

根据我的经验，这种过程不单单发生在 Etsy 身上，也发生在绝大多数公司身上。图 4-3 展示了这种产品开发过程的大致流程。

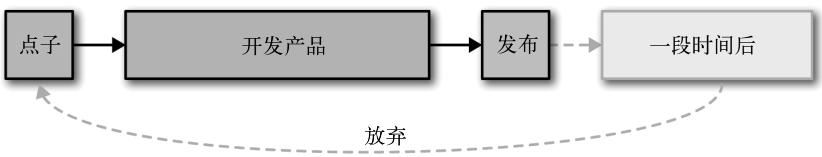


图 4-3：典型的产品发布过程，图片根据 Dan McKinely 的演讲而绘制

用这种方式做出成功产品的概率是很低的。更糟糕的是，有时候你甚至无法判断产品究竟是成功还是失败。假设你对网站进行了重新设计，一个星期后，激活用户就多了 10%，这一现象的原因可能是重新设计，但也可能是一些完全不相关的因素在起作用，比如网站的 Google 搜索的排名发生了变化。在这样的产品发布过程下，我们是无法找到确切原因的。

解决这种问题的方法之一就是进行 A/B 测试。和把新功能发布给所有用户不同，我们随机把用户分成 A 组和 B 组，其中 A 组是无法看到新功能的受控组，B 组是能够看到新功能的实验组。经过一段时间之后（这取决于花多长时间才能获得足够多的网站访问者，使得结果具备统计意义），我们就可以关注 B 组的指标是否和 A 组有所不同。如果不同的话，很可能就是新功能发挥了作用，因为它应该是两组之间唯一的变量。

如果发现新功能可以改善指标，我们就可以把它铺开提供给所有用户；否则就得抛弃它，重新开始。这样你就进入了一个产品的改善开发过程，如图 4-4 所示。

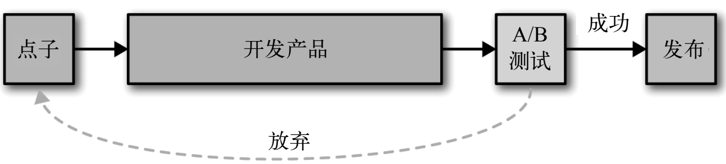


图 4-4：在后期使用 A/B 测试的典型产品发布过程。图标根据 Dan McKinely 演讲内容绘制

显而易见的是，如果我们在产品发布之前使用了这样的过程和 A/B 测试，很可能会发现产品的大部分功能要么对指标是有害的，要么就没有效果，这正是 Etsy 在开始进行 A/B 测试时发现的。我从 LinkedIn 和为本书去访谈的几乎所有创业公司身上都看到了一种产品开发模式。这些公司会花数月或数年的时间，耗资数百万美元去开发新产品，但很多产品做出来却无人问津，一两年之后就被抛弃了。

令产品失败的原因有许多，但其中之一就是 Tim Harford 所说的**上帝情结**（以为自己无所不能）。许多人，特别是专家，都相信自己只需要深入思考，就可以解决几乎所有问题。他们可以在纸上画出周密的产品点子、聪明的工程设计又或者是精美的图表和等式，然后等待成功的到来。只可惜，在大多数情况下，成功永远不会到来。那是因为我们所生活的世界无比复杂，通常我们面对的系统已经超出了任何个体的理解能力，比如自由市场经济、人的思想或者分布式计算机系统，影响这些系统的问题太过复杂，从单一原因入手无法得到解决。

> 我并不是想表达我们身处复杂的世界就无法解决复杂的问题。我们肯定是可以做到的，但我们应该谦逊地解决问题——抛弃上帝情结，采用实际可行的解决手段，我们也有具备了这种可行性的解决手段。现在，只要给我一个成功的复杂系统，我就可以给你一个通过尝试和犯错而得以进化的系统。
>
> ——Tim Harford，经济学家

我们需要的是进化，而不是聪明的设计。这意味着不要有上帝情结，要承认自己并不知道正确答案是什么。这也许是困难的，因为学校教育使我们习惯认为所有问题都有正确答案，只要深入思考就可以找到它。虽然我们在学校里遇到的简单、有约束条件、量身定做的问题确实是这样，但在商业领域遇到的问题是不会有简单、显而易见的解决方案的。一家又一家的公司已经发现，在这样的世界生存下来的唯一方法就是尽可能多地尝试，看看哪种方法可行。

> 在审视一些有远见的公司的发展历程时，我们总是惊讶于他们并不是依靠详尽的战略规划去做出最佳选择，往往都是依靠试验、尝试与犯错、投机主义以及——一点也不夸张地说——就是偶然。那些事后看似英明的策略，通常都是投机取巧地试验和“刻意的偶然事件”所产生的结果。
>
> ——Jim Collins、Jerry I. Porras，《基业长青》

要注意的是，尝试与犯错和盲目地猜测并不是一回事。我们仍然要尝试尽最大努力找出问题的原因，但也要承认有些假设可能是错误的，而找出那些错误的唯一方法就是反复试验。几个世纪以来，科学家们已经知道，进行反复试验的正确方法就是受控试验。

2. 通过受控试验实现数据驱动开发

做出了完整的功能或产品，但在进行了 A/B 测试之后才知道它其实没什么作用，这样的过程代价高昂且让人痛苦。那么，是否有方法可以避免这样的努力白白浪费呢？是的，就像我们无法消除测量中的各种不确定性，我们在做产品的时候也无法完全避免工作中的浪费——但我们可以减少这样的浪费。为此，需要在整个开发过程中利用数据和受控试验，如图 4-5 所示，应用迭代的方法论，而不是一开始就投入精力把整个产品做出来。

(1) 做一个 MVP。
(2) 对其进行 A/B 测试。
(3) 分析结果并做出下面三个决定中的一个。

a. 改善：测试得出的数字不错，足以证明我们能够进一步完善 MVP，回到步骤 1。
b. 发布：测试得出的数字非常好，并且产品已经完成，可以向所有人发布。
c. 放弃：测试得出的数字并不好，不足以证明应该继续工作，可以转到下一个点子上。

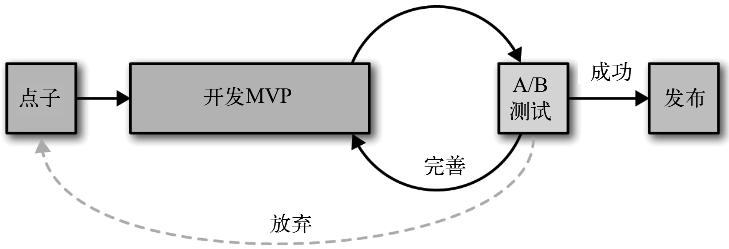


图 4-5：数据驱动开发，图表根据 Dan McKinely 演讲内容而绘制

我们应该迭代式地对 MVP 进行一些小投入以测试假设，从每次试验中收集数据。如果得到的数字印证了假设，就可以进一步投入，而不是在一开始就对产品进行大量投入。例如，在第一次迭代时，MVP 也许只是一个纸面上的原型，我们应用客户验证过程与真正的客户聊天，确定原型能否与客户产生共鸣（阅读 2.2.2 节了解更多信息）。如果客户的反馈不错，下一次迭代就可以是绿野仙踪式的 MVP（阅读 3.2.1 节了解更多信息）和 A/B 测试。如果 A/B 测试表明该 MVP 对我们的指标（特别是神奇数字）有正面影响，就可以做出更完整的原型，再运行一次 A/B 测试。可以持续这样的"实现 – 验证"循环，直至产品完成；或者，如果弄清楚了想法不可行，则放弃开发。但即便想法是行不通的，有了数据驱动过程，我们也可以更早得出结论，显著减少浪费的精力，实现所谓的速度制胜。

3. 数据驱动开发的优缺点

> "开发设计的时候要坚定你在做正确的事，阅读数据的时候要提醒自己可能会出错。"
>
> ——John Lilly，Greylock 合作人

数据驱动开发可以告诉我们所做的东西是否可行，但这取决于我们对数据的解读，只有深入解读才能理解可行或不可行的原因。数据驱动开发在比较不同选择时有极佳的表现，但前提是你要先想出这些选择。数据驱动开发是增量式地改进产品的完美方式，但也需要你尽巨大的努力，避免陷入局部瓶颈中。简而言之，数据驱动开发只有在我们把人的优势（例如创造力和洞察力）和计算机的优势（例如数据收集和测量）结合起来之后才有最佳的表现。我们使用数据是为决策制定的过程**提供信息**，而不是**替代**这样的过程。

除了为产品开发过程提供信息，我们也可以将数据作为构建**数据类产品**的要素。例如，LinkedIn 最被认可的功能之一就是"你可能认识的人"（People You May Know，PYMK），这是一个尝试预测用户在网站中还认识谁的推荐系统。PYMK 通过处理大量数据产生推荐信息，这些数据包括关系数据（例如，如果 Alice 认识 Bob，Bob 又认识 Carole，那么

Alice 可能也认识 Carole）、教育和工作数据（例如，如果 Alice 和 Bob 同一时间在同一学校或公司，他们就很可能相互认识）以及地理数据（例如，如果 Alice 和 Bob 都在同一个城市，他们就有可能相互认识）。该系统也会将用户的行为数据（例如用户是否点击了某一封推荐信）作为反馈提供给推荐引擎（即强化学习）。

数据类的产品也可能会有强大的独特作用，例如 LinkedIn 上超过一半的联系都是由 PYMK 带来的；Amazon 宣称它们有 35% 的产品销售来自它的推荐系统；Netflix 以其电影推荐系统而著称，部分因为它有很出色的推荐，部分因为它们举办了一个竞赛，该竞赛会对任何能够实现出更好系统的人提供一百万美元的奖励。

4.2 营销

到目前为止，我们已经讨论了构建出色产品的所有方法：需要有好的点子，需要制作低成本的 MVP，需要想出简单的设计，需要使用数据为决策提供依据。不幸的是，即便你想方设法做成了惊人的产品，但最出色的产品并不一定能够胜出。

在 20 世纪 90 年代末，TiVo 发明了第一台消费领域的摄像机（DVR），并且培养了一批非常忠诚的客户。这些忠实客户沉迷于该产品能够停止和回看直播电视节目，能够预先录制他们喜爱的节目。但是到了 2008 年，TiVo 在 DVR 领域的市场占有率仅有 6%，另外的 94% 被其他销售 DVR 的有线电视公司所占领。据说他们的产品并不如 TiVo（功能少、用户体验也差），但是有线电视公司却拥有出众的营销策略：他们把 DVR 作为用户必不可少的有线盒子的升级版提供给客户。

在 20 世纪 80 年代早期，微软的 DOS 操作系统在功能和用户友好程度上都不如 Apple 的操作系统，但 Apple 把它的软件做成专用的，只允许在 Apple 的硬件上发行，而微软则把操作系统的授权提供给任何购买的人，只要购买就可以安装。因此，许多计算机制造商，包括 IBM 和 IBM PC 的所有模仿者，都购买了微软的操作系统授权，DOS 操作系统一下子就覆盖了廉价的 PC 市场。到了 2000 年，微软把 DOS 替换为 Windows。虽然该系统是否能够媲美 Apple 的操作系统没有定论，但它却控制了 97% 的市场。

如果我们生活在一个可以获得完美信息的世界里，那么最好的产品总能胜出，但我们生活的世界并非如此。现今，许许多多吸引眼球的事物让我们目不暇接，客户根本无法看到所有产品，哪怕只是其中一部分。如果客户不知道你的存在，你的产品做得再好也没有意义。因此，现在并不是最出色的产品胜出，而是客户认为最出色的产品胜出。让客户觉察到你的产品并且影响他们对产品的感知，这就是所谓的营销。

“酒香不怕巷子深”的说法并不正确，因为产品不具备自我销售的能力。想要获得成功，你不仅要做产品，还要找到营销方法。

> 我们不妨把营销当作是产品设计最必不可少的一部分。如果你有了新发明，却没有销售它的有效方法，这就是糟糕的生意——不管你的产品多么出色。
>
> ——Peter Thiel，《从 0 到 1》

以下是创业公司最常见的 4 种营销渠道：

- 口口相传；
- 市场推广；
- 销售；
- 品牌化。

4.2.1 口口相传

传播产品信息最强大的方法就是不自己去传播，而是让你的客户去传播。没有公司可以承担所有市场推广手段的费用，所以每一家公司都要依赖某种形式的口碑营销。也就是说，让客户向不是客户的人推荐你的产品。有三件事可以让你通过口口相传的方式去提高营销效果：

- 做出更好的产品；
- 提供出色的客户服务；
- 让产品进入病毒式循环。

1. 做出更好的产品

尽管没有一种产品能够真正地自我销售，但你可以把一个产品做得好到让客户忍不住要谈论它，从而接近这一理想情况。例如，CrossFit 是一家 2000 年成立的健身公司，专为客户提供训练计划。现在，15 年过去了，它已经成为有史以来增长最快的运动项目之一，有超过 1000 万的 CrossFit 健身者在世界各地超过 10 000 家分店加入了训练（对比一下，麦当劳花了 33 年才达到 10 000 家分店）。CrossFit 快速增长的一个原因就是世上似乎没有其他可以与之竞争的健身课程了。CrossFit 并不向你承诺，每天在空调健身房里的昂贵健身器械上做个 5 分钟的练习就可以练出腹肌；CrossFitt 向你承诺的是，只要在普通的房间、车库、停车场，使用杠铃、壶铃、体操吊环、绳索、雪橇、拖拉机轮胎和大锤，通过大量高强度、不断变换的全身动作（其中糅合了跑步、举重、体操等各种健身运动），你就可以达到效果。它和 CrossFit 健身者所嗤之以鼻的健身课程是完全不同的，或者就像笑话讲的：“你怎么知道谁加入了 CrossFit ？别担心，训练者自己会告诉你的。”

从中可以看到，CrossFit 之所以值得在此讨论，正是因为它有**独到之处**。它并不具备每一种可能有的功能（作为健身课程，CrossFit 有大量的空白和不足），但它在少数功能上却做得异常出色（阅读 3.2.2 节了解更多信息）。把注意力放在差异性上，不仅对实现出色的 MVP 是非常重要的，而且也可以提升客户关注产品的概率，因为它与众不同。

Seth Godin 在 TED 演讲“How to Get Your Ideas to Spread”（如何传播你的点子）中，对这种思路做了一个出色的比喻。想象你正在开车，看到路边有一头牛，你会不会停下来观察它呢？很可能是不会的，因为你之前已经看到很多次牛了，所以你只会接着开车。但如果你看到紫色的牛，几乎都会把车停下来拍照。为什么？因为紫色的牛是**引人注目的**。对产品来说也是一样的，各种可供选择的产品和推广信息会让客户感到目不暇接，所以你让他们关注你的唯一方法（也是最重要的方法）就是——让他们的朋友关注你，这就取决于你是不是做了一些引人注目的事。你必须有所不同，值得人们去谈论。

2. 提供出色的客户服务

不管付出多么大努力去做出让人惊讶的产品，都不可能把每件事情都做对。客户会遇到问题，会遭遇 bug，会碰到个别的状况并要求你提供新功能，这就是客户服务的由来。把

客户服务放在“营销”这一节讨论可能有点奇怪，但许多公司都发现，如果客户服务做得特别出色，也可以成为你与众不同的地方，获得强大的口头宣传效果。

> 多少年来，Zappos 增长的第一驱动因素就是回头客和口口相传。我们的理念就是把本应花在付费广告上的大部分资金投入到客户服务和客户体验中，让我们的客户通过口口相传帮助我们进行市场推广。
>
> 当我参加有关市场推广和品牌宣传的会议，听到一些公司谈到客户每天都被成千上万的广告信息轰炸的时候，我自己觉得这是有点可笑的，因为公司和广告商之间经常会进行许多讨论，谈论如何让他们的信息脱颖而出。近来对“社交媒体”和“整合营销”的讨论越来越热闹，虽然电话听起来不怎么吸引人，也没什么技术含量，但我们相信电话是目前为止最好的品牌宣传设备。让客户专心致志地听五到十分钟，如果能进行正确的交互，我们发现客户会在很长时间内记住这次经历并告诉他（或者她）的朋友。
>
> ——Tony Hsieh，《回头客战略》

如果你真的想在客户服务方面做得非同凡响，独立、外包的客户服务部门通常是不够的。Zappos 让每一个雇员都参与到客户服务中（阅读 11.2.5 节了解更多信息）。同样地，Stripe 的每一位工程师，甚至他们的创始人，都会每两周轮流去做客户支持工作。如果工程师都跑去做客户服务，还可能让公司发展壮大吗？是的，我们以前讨论过，在公司成立的早期，做一些无法规模化的事情是最好不过的。让每个人都参与到客户服务中，对公司的扩大发展确有惊人的作用，因为这么做不仅可以让忠诚的客户传播你的产品，还可以让这些写代码的人也感受到使用产品的客户的痛苦，帮助你做出更好的产品。例如，KAYAK 的创始人 Paul English 在工程师的席位中间装了一条客户支持电话线。人们经常会问他：“为什么你要让拿着高薪的工程师去回答客户来电？”他的回答是：“要是电话隔三岔五说的是同样的问题，工程师们就会停下手头的工作，修复 bug，免得反映这个问题的电话再打来。”

我们在上一章已经谈论过，应用客户开发过程，我们可以走出办公室，经常性地在真实的客户身上验证假设。客户服务给我们带来的好处也是一样的，唯一的不同就是让客户来找你。我们要确保在产品的显眼之处放上接受反馈的邮件地址或者电话号码，或者使用像 ZenDesk、Groove 和 Get Satisfaction 这样的工具，管理我们与用户的沟通交流，让用户更容易地找到我们。

3. 让产品进入病毒式循环

近来许多人都在谈论如何使用“病毒营销策略”，但现实中却没有这样的东西。在所有社交网络上出现的病毒式传播的博客文章或视频并不是一种市场推广策略，而是纯属好运。这是你预料不到的，你也无法控制能看到它的受众，也不能把它变成一种可持续的营销策略。病毒式传播不仅仅是另一种形式的口口相传，如果你想突破我们前面讨论的实现途径（做出更好的产品、提供出色的客户服务），更进一步激发人们对你的产品进行口头宣传，需要的不是一种“病毒式的市场推广策略”，而是让产品进入一种病毒式循环。

病毒式循环是产品的一个特性，它为当前用户提供吸收新用户的激励。反过来，新用户又有动机去邀请更多的用户，就像病毒一样传播你的产品。在 20 世纪 90 年代末期，PayPal 的现有用户每推荐一个朋友，PayPal 就提供 10 美元的奖励，而新用户一注册就

可以得到 10 美元。他们赌的就是，只要用户注册，就离不开这个服务，就会带回足够的钱，进而覆盖获得每个用户的 20 美元的成本。这冒着巨大的风险，可能不是很多公司能够复制的策略，但对 PayPal 是有效的。该举措推动 PayPal 的用户每天增长 7%~10%，到服务结束时，已经累积增长了超过 1 亿用户。

有些病毒式循环根本就不需要用户的介入。例如，Hotmail 在 1996 年第一次推出的时候，是世界上最早的基于 Web 的免费邮件服务之一，但它们还是在想方设法要把这一信息传递给大量用户。他们决定试试病毒式的策略：每次用户发送邮件的时候，Hotmail 会自动在邮件的底部添加一个签名，其中包含一个“在 Hotmail 获取你的免费邮件”的链接。只要有人收到来自 Hotmail 用户的邮件，他们就会知道：发送者，通常就是他们所信任的人，即用户；该服务正在运转中；该服务是免费的。该签名一上线，Hotmail 的就开始飞速发展。一开始一天增加几千个用户，6 个月内增加了 100 万，此后几周又增加了 200 万，一直持续下去。

最强有力的病毒式循环应该是产品本身使用功能的一部分。如果一种产品除非被使用，否则无法让客户从中体会到价值（比如电话、视频聊天或短信），那么邀请新客户就会成为使用该产品的内在要求，这样你就有机会获得病毒式的快速增长。但这里有一个问题：你怎么让最初的用户去注册呢？在还没人有电话的时候，电话不知打给谁的时候，怎么说服别人买电话呢？这就是所谓的**冷启动问题**。像电话这样的产品遵循的是梅特卡夫定律：产品的价值与用户数量的平方（n^2）成正比。也就是说，这样的产品在开始时会很困难（$0^2=0$），但一旦让球滚动起来，就会获得强大的**网络效应**，即每一个新用户都会显著地增加网络的价值，从而吸引更多的新用户，又会再次提升网络价值，如此循环往复。

社交网络就是内生式病毒增长和网络效应之强大的一个经典例子。社交网络的全部意义就是与他人的联系，所以发送邀请就是使用该产品的内在需求，由此才能引起爆发性的增长。在 2014 年年末，LinkedIn 有 3.47 亿会员，Facebook 有将近 14 亿会员。这些社交网络又是如何解决冷启动问题的呢？首先，他们把用户邀请非会员变得很简单，允许用户从邮件、手机和其他现有的网络中导入联系人（病毒式的软件产品在这一点更容易做到，因为新用户不需要购买任何实体的东西，比如手机）。其次，他们在网络发展壮大之前，就已经向用户提供了有价值的东西。例如，LinkedIn 即便在会员还很少的时候，就已经是可以存放简历的公开场所，可以让潜在雇主和商业伙伴发现你，这点对会员已经很有用了。

这些例子应该会驱散你对病毒式循环的少部分错误看法。首先，这不是免费的，在产品中实现病毒式循环总要付出成本，如果病毒机制并不是用户体验的内在固有部分，你也许要为每一个新的用户付出真金白银，就像 PayPal 的例子一样。其次，虽然几乎所有的产品都可以从口口相传中得到好处，但并不是每一种类型的产品都能够引入病毒式循环。下面是几个你应该要考虑的问题。

- 用户如何才能生成可以送达到其他用户的内容？
- 如何做到用户联系的人越多，体验就越好？
- 如何让用户在向非用户伸出手时，就可以获得好处？

——Adam Nash，Wealthfront 主席、CEO

如果产品内在就有社交属性——产品被多人协作使用，比如社交网络、文件共享服务或

者支付应用——回答上面的问题一般会比较简单。如果不是的话，要实现可持续的病毒式循环有可能是很困难的。想要了解这么做在时间上的投入是否值得，可以进行一些估算，看看可以得到什么样的回报。

第一步就是评估你的病毒系数。**病毒系数**（又称为病毒因子）是一个数字，它回答的是以下问题。

> *假设今天我获得了一个新客户，经过 N 天之后他可以给我带来多少新客户？*
>
> ——Adam Nash，Wealthfront 主席、CEO

数字 N 表示业务的合理**循环时间**，即通常情况下一个新客户发出邀请且接收者做出回应所花的时间。以 Facebook 这样的产品为例，一种合理的推测应该是 $N = 1$ 天，因为一个新用户通常会在注册之后立即发送出他的所有邀请，这些邀请都是通过 email 和手机通知发送出去的，接收者很可能在同一天内就会回应。另一方面，像 SlideShare 这样的产品，新用户也许会在注册之后立即提交一张幻灯片并分享给他的朋友，但这些朋友很可能要等到自己也有幻灯片要分享的时候才会注册 SlideShare，可能会是数月之后，所以可能会是 N=180 天。

要计算病毒系数（K），需要用每 N 天的用户发送邀请数（I）去乘以这些邀请的平均转化率（C）。对于 I 而言，就是当前一个用户执行了多少次才吸引来新用户的某个动作，这种动作就好比在社交网络上发送一个邀请；对于 C 而言，就是邀请被接受的百分比是多少。

$$K = I \times C$$

例如，假设你今天发布产品，有 1000 个人注册，通过查看指标发现，这 1000 个用户在注册后不久即发送了 5000 个邀请，或者平均下来每个用户会邀请 5 个新用户，即 $I = 5$。这些邀请在最开始的几天会获得许多点击，在大概一周左右就会降到 0，所以你的循环时间就是 $N = 7$ 天。在这一周的结尾，你会发现这些邀请带来了 500 个新用户前来注册，所以转化率就是 $C = 500 / 5000 = 0.1$。由此可以得出病毒系数 $K = I \times C = 5 \times 0.1 = 0.5$。假设这些数字不变，你在一周后就可以获得 $1000 \times 0.5 = 500$ 个新用户，第二周就可以获得 $500 \times 0.5 = 250$ 个新用户，以此类推：

$$1000 + (1000 \times 0.5) + (1000 \times 0.5^2) + (1000 \times 0.5^3) + \cdots$$

有了病毒系数 K 和循环时间 N，就可以计算产品发布 T 天之后的用户数：

$$\text{Users}(T) = \sum_{i=0}^{T/N} \text{Users}(0) \times K^i$$

如果你还记得高中数学，就知道这是一个几何级数，该几何级数的前 x 项的和可以表示为：

$$\text{Users}(0) \times \frac{1 - K^x}{1 - K}$$

如果 $K < 1.0$，随着 x 的增长，K^x 也会随之增长，该等式可以简化为：

$$\text{Users}(0) \times \frac{1}{1 - K}$$

如果把 $K = 0.5$ 代入病毒系数中，就会发现这个几何级数收敛于开始时用户数的两倍。同样，如果我们代入 $K = 0.67$，就会得到用户数的三倍，$K = 0.75$ 就是用户数的四倍，等等。这意味着病毒系数在 0~1 时可以看作是一个固定的乘数，如图 4-6 所示。当我们把病毒式增长和其他可持续的营销策略结合起来时，它就成为扩大产品用户数的强大手段。

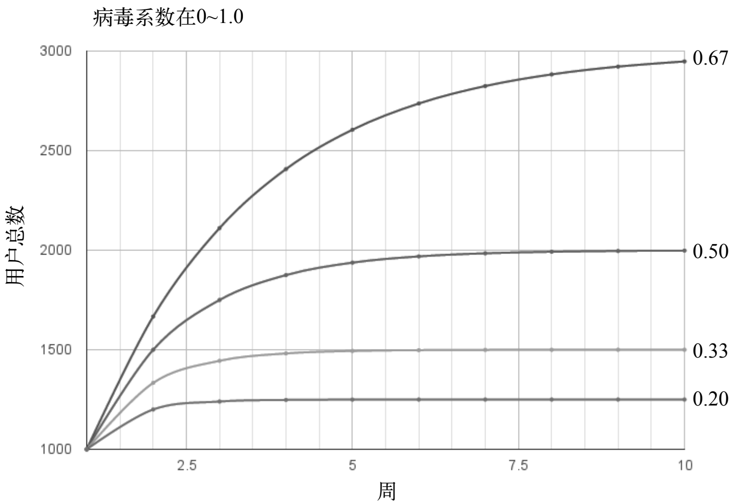


图 4-6：从 1000 个用户开始，病毒系数在 0~1.0 的用户增长情况

但如果病毒系数 ⩾ 1.0，情况又如何呢？例如，如果病毒系数是 1.5，最初的 1000 个用户将在第二周带来（1000 × 1.5）= 1500 名用户。这些用户又会在下一周带来（1500 × 1.5）= 2250 名用户，接下来一周又是 3375 名用户，以此类推。如果这一模式继续下去，得到的就是指数级的增长，用不了多长时间，地球上的每个人都在使用你的产品。很明显，这是不现实的。实际当中，没有一种产品可以把病毒系数维持在 1.0 以上超过一小段时间，大多数产品的系数都要小得多。

> 真正的病毒式增长极其罕见，我花了一段时间才领会这一点：只有非常少数的产品能够让病毒系数在一段时间内都超过 1。但如果我们不应该下赌注赌病毒因子会大于 1，我们在模型中应该使用什么数值呢？
>
> 在和其他创业者、投资者和增长黑客（growth hackers）[2] 的探讨中，我学到了这一点：对于消费性互联网产品，可持续的病毒系数在 0.15~0.25 就不错了，在 0.7 左右就已经很突出了。
>
> ——Rahul Vohra，Rapportive 联合创始人

既然极少有产品可以把病毒因子维持在 1.0 以上，我们就不能仅仅依赖于公关活动和病毒式增长，还需要其他可持续的营销机制。例如，假设产品通过 Google 搜索一个星期可以获得 10 000 个访问者（阅读 4.2.2 节了解更多内容），这些访问者中有 500 人去注册。看

注 2：“增长黑客”这一概念近来兴起于美国互联网创业圈，最早是由互联网创业者 Sean Ellis 提出。增长黑客是介于技术和市场之间的新型团队角色，主要依靠技术和数据的力量来达成各种营销目标。——译者注

看图 4-7 就可以了解，把通过公共关系活动获得的初期的 1000 用户和搜索带来的 500 个注册用户，放在各种病毒系数的作用下，会有什么样的结果。

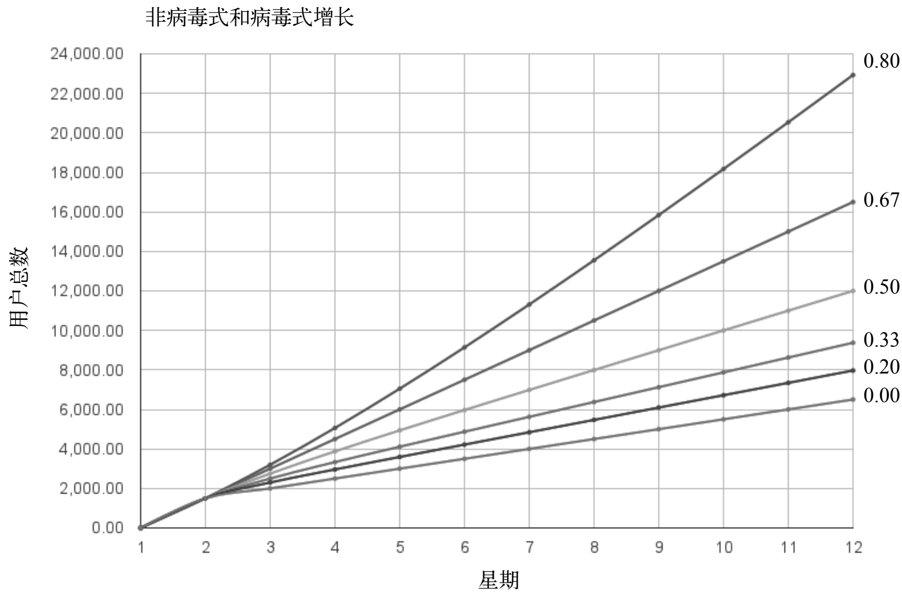


图 4-7：病毒式和非病毒式增长的结合

如果产品增长完全不是病毒式的，在 12 周之后，只有 6500 多名用户；如果病毒系数是 0.5，大概可以获得 12 000 名用户；如果真的取得了突破，病毒系数达到 0.8，将获得近 23 000 名用户。注意，这一计算并没有考虑许多因素，比如用户留存（即每一周有百分之多少的用户停止使用产品）以及病毒系数在此期间的变化方式（即这一计算假设新用户一加入进来就发送邀请，但此后不会再发）。如果读者想了解关于病毒式增长建模方式更完整的讨论，或者想看看在计算过程中可以使用的各种便捷的电子表格，可以查看文章“How to Model Viral Grawth: The Hybird Model”。

4.2.2　市场推广

现在，我们把关注点从口口相传（即新客户通过现有客户去发现你的产品）转到市场推广（即新客户直接从你这里发现产品信息）上。对一个产品做市场推广有许多方式，我们仅仅介绍创业公司最常用的一些：

- 广告；
- 公共关系和媒体；
- email；
- SEO（搜索引擎优化）；
- 社交媒体；
- 集客式营销。

1. 广告

广告几乎就是市场推广的代名词。要利用这一手段，我们必须找出潜在客户的关注点，付钱将产品的信息放在那里。广告好的一面是它是有效的，而不好的一面则是所有人都知道它是有效的。仅美国一地，广告就是一个价值 2200 亿美元的产业。不管你到什么地方，都会有广告在争夺你的注意力。电视上、广播中、飞机上、广告牌上、公共汽车的每一面都有商业广告；在每部电影、每个节目中，也都会植入产品，还有名人代言推荐的现象；在所有报纸、杂志、体育场、电影院和音乐厅也都有广告；在公园长椅和人行道上、T 恤和帽子上、标签和贴纸上也会有广告。当然，电脑上、手机上也会有广告。这些广告有横幅广告、浮动广告、搜索广告、新闻推送中的赞助商内容、手机广告，以及我们在观看视频或阅读文章时不得不慢慢等待的各种插播广告。

如果我们打算在广告宣传上投入资金，就必须严密跟踪用户获取渠道，这样才能判断广告是否有效（阅读 4.1.1 节了解更多信息）。如果利用的是在线广告，这样的跟踪会容易实现，通常可以判断出用户是否是通过广告点击过来的。如果用户是通过广告而来，我们也清楚花费的广告成本以及用户带来的回报。虽然跟踪电视商业广告这样的传统广告会更加困难，但也并非是不可能的，其中一种方法就是利用促销代码。例如，在每次广告宣传活动中，可以告诉用户必须输入特定的促销代码才能获得折扣，这就相当于只要激励用户在你这里登记一下，你就可以跟踪用户是从什么渠道过来的。另一种办法就是使用客户调查，例如 TripAdvisor 发布了一系列电视广告之后，就向用户发出调查，看看这个活动覆盖了多少人，以及活动的效果如何。

尽管调查和促销代码都不像在线跟踪一样精确，但我们的目标并不是为了得到完美的数据，仅仅是降低一些不确定性，只要能够测量出用户的来源以及是什么样的用户就行了。因为我们不仅仅要确定广告已经到达了受众，还要确定它到达的是不是正确的受众，这一点在电视和广告牌这样的广播式媒介上，通常需要一定的技巧去处理。

2. 公共关系与媒体

除了广告宣传之外，公共关系（public relations，PR）是另一种可以把产品信息呈现在大量受众面前的方法。为此，必须和电视或电影行业从业者、媒体记者、博主以及名人建立起关系。当然，如果你做的是夺人眼球的东西（或者一些相当糟糕的东西），有时候即便你没有请他们，他们也会谈论你。

对于大多数创业公司而言，PR 事件是不可预测的，偶尔还是负面的，也极少会是可持续的推广策略，这些事件可以很好地产生流量峰值，但在几天之后，峰值通常会平息下来，我们又需要从头再来。

3. email

如果方法得当，email 营销可以取得令人难以置信的效果；如果方法不得当，就不过是一堆垃圾邮件而已。错误地利用 email 就是**直接营销**，就像购买一份 email 清单，然后把大量邮件发给一堆从来没有听说过你的陌生人。这并不是一种可持续的策略，因为这些邮件的点击率（click-through rate，CTR）是极低的（通常仅有百分之零点几），而且你的 email 很快就会被大多数主要的 email 运营商归为垃圾邮件，对公司名声也有所损害，所以只能在少数活动中发送这样的邮件。

更好地使用 email 的方法是创建一份选择接受你产品信息的用户 email 列表。例如，可以

让用户在宣传页面上注册订阅新闻（阅读 3.2 节了解更多信息）。客户通常要花很长一段时间才能决定购买，对产品感兴趣之前也许要一次又一次地去了解产品。所以，偶尔发送一份带有用信息的提醒邮件，可能是把感兴趣的人转变成付费客户的好办法。与直接营销相比，这种方法的 CTR 会稍高一些（大概 1%），email 被归为垃圾邮件的可能性也更小。

使用 email 的最好方式就是根据用户所关心的活动或事件相应地发送个性化的邮件。例如，每当有人在 Facebook 的照片里把你标注出来的时候，你就会收到一封邮件。我不知道你会怎么做，但我自己对这种邮件的 CTR 几乎是 100%，因为我得确定那不是一张我会后悔的照片。2011 年，LinkedIn 推出了“一年回顾”邮件，这是一封年度邮件，展示了过去这一年你所有在职业上有重大变化的同事的照片。这封邮件的 CTR 是极大的，因为许多用户会不止一次点击邮件中的链接（CTR > 100%），看看谁找到了新工作或者得到了晋升。

还有一点也很重要，那就是我们要认识到 email 并不是获取新用户（用户获取）的良好手段，但它是吸引现有用户的最佳手段之一（用户激活、留存、推荐和收益）。我们可以发送的 email 有很多类型，比如欢迎邮件、入门邮件、再次接触邮件以及推荐活动。大家可以看看 sendwithus 网站提供的“如何像创业公司一样发送 email”教程，了解更多相关内容。

4. SEO

搜索引擎优化（search engine optimization，SEO）就是对网站进行优化，让网站在搜索结果中有较高的排名。Google 在 2012 年经历了 1.2 万亿次搜索，许多大公司都以这样的巨大查询量为基础对市场进行了划分，建立起自己的商业模式。如果你的产品拥有大量独特而有价值的内容，比如用户评论（如 TripAdvisor）、论坛（如 Reddit）、问答（如 Stack Overflow）或者参考材料（如 Wikipedia），你就可以利用 SEO，某种程度上不用投入资金就可以带来数千乃至数百万的页面浏览。它不仅可以给你带来大量可持续程度较高的流量，而且这些流量通常都会有很好的转化率，因为它代表的这些人所搜索的内容正是你拥有的。

其中的问题在于，Google 和其他搜索引擎所使用的排名算法是保密的。Google 发布过一份 SEO 初学者指南，目前也有许多关于 SEO 秘诀和技巧的资源 [3]，但无法保证这些技巧和排序算法与搜索引擎实际的工作方式是对应的。更难的是，Google 每年对排序算法的修改超过 500 次，虽然大多数改变都比较小，但有一些还是会对你的排名结果带来显著的影响 [4]。这就意味着 SEO 并不是完全免费的：我们必须预先投入工作对网站进行优化，也必须跟上排名算法的变化，让网站处于结果页面的上方。当然，我们还得和其他尝试做相同事情的网站进行竞争。

SEO 的好处在于，排名算法一般都会对你做对的事给予回报（不管你做不做 SEO）。例如，我们可以做的最重要的“优化”就是让网站拥有大量高质量的内容——做出更出色的产品（阅读 4.2.1 节了解更多内容）。如果拥有高质量的内容，其他许多网站都会链接到你这里，当用户点击你的网站时，他们不会立即离开，这两个因素都会增加你的页面排名。其他一些优化手段也是有用的，比如调整页面头部的标题、URL、域名和 meta 标签，但影响几乎都不及你做出出色的产品。

注 3：Moz 网站提供了一份相当不错的 SEO 最佳实践指南，“On Page Ranking Factors”。

注 4：Moz 网站维护了一份 Google 排名算法的“变化历史”，“Google Algorithm Change History”。

5. 社交媒体

许多公司都转而使用社交媒体进行推广，他们有很好的理由，因为社交网络拥有众多具有较高参与度的受众。在 Facebook、Twitter、LinkedIn、Instagram、Pinterest 和类似的网站上培养一批追随者，是吸引已有用户的好策略——这就像更现代化的 email 新闻订阅。更妙的是，我们可以通过社交媒体与用户进行单独的互动，使之成为一种高效的客户服务手段。唯一不是特别好的就是用户获取，当你在 Twitter 或 Facebook 上分享东西时，现有的关注者会看到它，但除非有人转发，否则新用户是看不到的。你的一些内容偶尔可能会得到大量的转发和“病毒式扩散”，使得产品可以呈现在很多新用户面前。然而，这就有点像公关活动，只是一次性的提升，而且太不可预料，不能以此作为一种可持续的用户获得形式。

6. 集客式营销

集客式营销就是利用客户觉得有价值的内容去引起客户的关注，而不是像广告那样去购买客户的关注。我们可以把它想作用蜜罐吸引客户过来（**集客营销**），而不是用喇叭把你的营销信息发送出去（**推播式营销**）。蜜罐可以是博客、播客、视频、图书或一组开源工具等形式。我们通常把它和 SEO 以及社交媒体分享结合起来，帮助用户找到这些内容。集客式营销背后的关键理念就是不要尝试把东西卖给客户，而是尝试去教育他们。

> 教导客户，就可以和客户建立起传统市场营销策略所无法获得的纽带。通过杂志或网上横幅广告去购买人们的关注力是一方面，如果教育他们，赢得他们的忠诚，建立起完全不同的关系，他们就会更加信任你、更加尊重你。即便他们不用你的产品，仍然会成为你的粉丝。
>
> ——Jason Fried、David Heinemeier Hansson，《重来》

对创业公司来说，集客式营销是特别有效的策略，因为他们在广告预算方面无法与大公司竞争，但他们可以产生有价值的内容。sendwithus 就是一个很好的例子，他们运营一个关于 email 营销技巧的博客，发布了一份“如何像创业公司一样发送 email”的免费的、全面的教程。他们还免费赠送给用户许多 email 工具（例如模板、组件和布局工具）。如果你正在搜索有关发送 email 的帮助，比起一些广告或明显是在向你销售产品的信息，你会更倾向于点击一些免费而有价值的内容。你在阅读 sendwithus 博客文章和使用其工具的过程中，脑海中就慢慢在 sendwithus 和 email 之间建立起了一种关联。你会开始把他们当作是 email 方面的专家，只要遇到问题，就会想到他们。于是，如果有一天你需要付费的 email 产品，你就更有可能成为他们的客户。

4.2.3 销售

市场推广是让客户上门（用户获取），销售则是和客户达成交易，让客户购买（从用户处获取收益）。如果你的产品是“自助式的”，比如用户可以输入信用卡进行购买的网站，你的销售和营销过程在整体上都是一样的。但是，也有许多类型产品的销售是需要有人介入的——整个过程需要有销售人员在场，回答客户的问题，充分讨论合同的细节。在美国，粗略估算有 1400 万人从事销售行业，占人口比例 5% 左右。销售之所以是如此流行的职业，是因为销售是大多数生意成功的基础，甚至从更普遍的意义看，销售是人生中多数事情成功的基础。

> 甚至连商人都低估了销售的重要性，而最根本的原因就是在这个背后由销售驱动的世界中，各个领域、各个层次的人整体上都在努力隐瞒这一点。
>
> ——Peter Thiel，《从 0 到 1》

就像所有职业都会涉及销售，公司的 CEO 也是如此，从许多方面看，他就是个销售人员，把公司的愿景销售给客户、投资者、股东和雇员。如果你从事市场营销方面的岗位或者从政，大部分时间也都是在销售。即便你是程序员，每次应聘工作或者面对职位进行谈判时，又或者说服你的团队采用新技术时，其实你也变成了销售人员。当然，这些工作的头衔中都没有“销售”二字，因为它属于我们都在玩、却不能去承认的一个游戏，否则就玩不下去了。如果你承认自己是在和别人调情，你可能就没法约会；如果你承认你正在设法卖东西，可能就没法达成交易。没人愿意被销售，但所有人要买东西，这才让销售成为如此困难的一个职业。

> 你是怎么学习销售的？说服别人使用你的产品，就像影片《华尔街之狼》说的那样：试着卖给我这只钢笔。认真点，试试看。你会怎么做？你要怎么样才能让这只钢笔变得对我很重要？只要你学会了，你就知道如何销售了。
>
> ——Matthew Shoup，NerdWallet 高级技术官

如果你建立了一家创业公司，就开始销售你的产品，即便（或者特别是）你的职业并不是销售人员，这都是一个很有价值的练习。在你雇用销售团队之前，在你投入大量金钱进行市场推广之前，你应该先走出来，亲自和客户交谈，试着把你的产品卖给他（阅读 2.2.2 节了解更多信息）。只有你自己卖出去少量东西之后，你才能从中了解到什么东西对客户是最重要的，什么样的销售策略行得通，这时才应该把雇用独立销售团队的事情放在心上（阅读 3.2.4 节了解更多信息）。

一旦到达了这个阶段，所需的销售团队的类型在很大程度上就取决于你的产品了。大致来讲，销售分三种类型。

(1) **自动化销售**是自助式的系统，客户无须和人交谈就可以完成购买，比如 Amazon 网站上的结账页面。
(2) **内部销售**就是销售人员在他们雇主的工作场所完成大部分销售。常见的一种内部销售就是在商店或经销商处工作的销售人员通过柜台去销售产品，比如 Apple 零售店中的员工。另一种常见的类型是在办公室工作的销售人员，通过电话、email、聊天工具和网络会议等方式销售产品。许多软件即服务（Software as a Service，SaaS）产品，比如 SalesForce，就允许客户注册在线产品的试用版或基本版（自动化销售），如果想要功能更强的版本，就可以通过电话或 email 和销售代表接触（内部销售）。
(3) **外部销售**是指销售人员在客户的工作场所完成他们的大部分工作。他们与预约客户安排个人见面，为客户提供现场演示，大部分时间都花在拜访客户和与利益相关者的直接交谈中。

自动化销售是最具扩展性和最具经济效益的选择，但通常只适用于低价产品（少于 1000 美元）。外部销售成本要高得多，扩展性也要更小——你必须雇用销售团队，支付他们差旅费，每个销售人员一次只能应对一名客户——但是这种针对个人的服务却适合销售更加昂贵的产品，这样的产品是不会有人通过网上结账页面去订购的（10 万美元以上）。内部销售介于两者之间，它比自动化销售需要付出更多成本，但仍不及外部销售团队，商

店中的每一名销售人员或者办公室打电话的人，每一天都可以和许许多多的客户进行交互。和客户之间的这种个人互动使得内部销售团队比自动化销售更能达成较大的交易，但仍然无法与外部销售团队相比（内部销售团队的规模通常就是在 1000~10 万美元之间）。

4.2.4 品牌化

我之前提过，并不是最出色的产品胜出，而是客户认为最出色的产品胜出。客户如何看待你的公司——如果他们想到的是你的品牌，那么尝试对这种感知进行影响的行为就称为**品牌化**。品牌化并不是一种单独的策略或者市场推广活动，而是你与客户交互的各种方式的总和：公司 logo 的外观、宣传口号的内容、广告中所展现的公司形象、集客式营销中提供了什么样的专业知识、网站的外观、商务名片的设计、销售团队所使用的策略以及客户服务中对待客户的方式。就像产品的差异性可以区分你的产品和其他公司的产品，公司品牌也要和其他公司有所不同。

红牛生产的是含糖的咖啡因苏打饮料，但它的品牌与同类公司是完全不同的。例如，在其网站上，如图 4-8 所示，展示的是一个骑越野摩托车比赛的人、舞会上的大学生，还有从山上跳下的人。红牛运营着一个红牛 TV 频道，播出定点跳伞、跑酷、冰山攀岩、激流划艇这样的极限运动；它拥有若干体育团队，包括纽约红牛（足球）、英菲尼迪红牛车队（一级方程式）、Team Red Bull（全美赛车协会）；它也会赞助一些活动，比如红牛公路速降赛（极限自行车速降竞赛）、红牛极限摩托车大赛（在斗牛场举办的自由式摩托车特技表演竞赛）、红牛平流层极限跳伞（太空跳伞项目，包括空中造型跳伞运动员 Felix Baumgartner 也参加了，他曾从 38 公里高自由落下，下降速度超过了每小时 1280 公里）。所以只要一想到红牛，你想到的不是饮料，而是极限运动——这就是它的品牌。

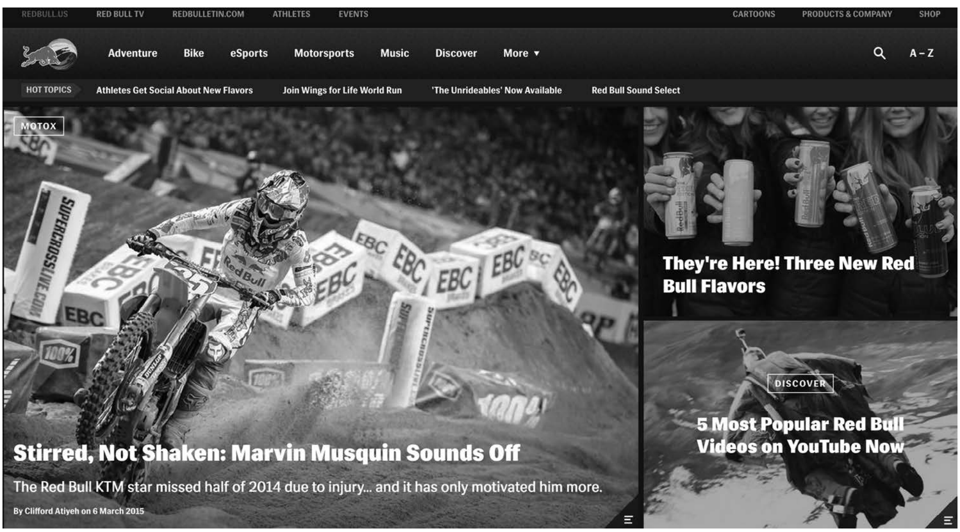


图 4-8：红牛网站

需要注意的是，品牌的建立与具体的产品关系不大，它关乎的是情感和信念，关乎公司为何存在而不是公司是做什么的（阅读 9.2 节了解更多信息）。例如，耐克的广告活动并没有介绍鞋子和空气鞋底，而是在表达对伟大运动员和伟大运动的尊敬。同样，Apple 最

成功的广告之一 ——“不同凡想”（Think Different）并不是在介绍电脑或 CPU 速度，也不是在介绍 Apple 为什么比微软更出色，而是在回答“Apple 是谁”以及“它代表什么”的问题。

> 献给疯狂的人、不合时宜的人、叛逆者和麻烦制造者。他们不喜欢规则，他们不尊重现状。你可以响应他们或否定他们，可以颂扬他们或诋毁他们，而你唯一不能做的是忽视他们，因为他们改变事物，他们推动人类前进。当有些人把他们视为疯子时，我们看到的是天才。因为疯狂到认为自己可以改变世界的人，就是真正改变世界的人。
>
> ——Apple，“不同凡想”

只用了“think different”两个词，就能够让你准确地知道 Apple 是做什么的，以及你为什么应该关注它。精心制作出这样一条清晰、引人注目的广告词并不容易，但正所谓文案是产品设计最重要的因素（阅读 3.1.3 节了解更多信息），广告词也是市场推广的核心。产品的宣传口号就是一个很好的例子，它必须能够一下子抓住人们的注意力，让别人知道你的产品有什么不同，而且必须简洁明了。例如，我们看看最初 iPod 的口号。

> 装在口袋里的 1000 首歌。
>
> ——iPod 最初的宣传口号

在 iPod 诞生的年代，大多数人要携带他们所收藏的音乐只能带着大量的 CD 包，每张 CD 中存放大概 12 首歌。而能够存放 1000 首歌的、可以放进口袋里的音乐播放器这一想法，就足以吸引全世界的关注。

某种意义上，你的品牌就是你要改变客户生活的承诺：如果你选择了我们公司，这就是你将会实现的。红牛承诺让你有精力去从事极限运动，Apple 承诺为你提供“不同凡想”的技术。注意，这样的承诺并不是产品所能做到的事情（功能），而是客户可以使用你的产品做到的事情（受益）。这是一个至关重要的差别，如图 4-9 所示。一旦你深刻地理解了这一点，就会发现想出效果出众的广告词会变得容易一些。

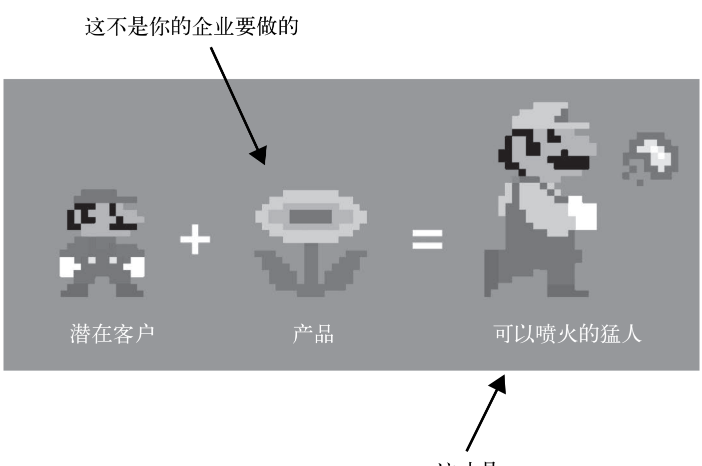


图 4-9：功能与受益的对比（图片由 Samuel Hulick 提供）

4.3　小结

Y Combinator 的座右铭就是“做人们想要的东西”（make something people want）。这四个简单的单词可以表达出创立成功的创业公司必须了解的几乎一切。前面的两章讨论了如何想出点子，以及如何以此设计出基本的产品。这两点对应的就是“做人们想要的东西”这句话中的“做”和“东西”。这一章分别从“数据”和“营销”这两点去讨论这句座右铭的另外两个单词——“人们”和“想要”。

> 如果有数据，那就一起参考数据。如果只有观点，那就听我的好了。
>
> ——Jim Barksdale，Netscape 前 CEO

数据能验证“我们所做的是**人们**想要的东西”这一点。我们所做的几乎所有决定都可以通过测量和降低不确定性而得到改进。我们可以严密跟踪海盗指标（AARRR）：获取、激活、留存、推荐和收益，从而掌握产品各个方面的情况。通过定义“神奇数字”并用它去安排所有项目的优先次序，可以让整个团队为着同样的使命去努力。最重要的是，数据不仅可以跟踪过去决定的进展情况，还可以为未来的决策提供依据。我们应该把上帝情结放到一边，使用 A/B 测试去评估人们真正想要的东西。

> 请把“完美的产品”想作是捕鼠器。你想要捉住老鼠，是因为你有完美无缺的捕鼠器，但即便是完美的产品，你也得把老鼠吸引过来。所以，要让捕鼠器成功发挥作用，真正要做的就是考虑诱饵和位置。我们必须用对食物去吸引老鼠，也必须把这个陷阱放到正确的位置——倚着墙，这就是你了解到的老鼠活动的地方。这就是位置和诱饵的作用。
>
> ——Matthew Shoup，NerdWallet 高级技术官

营销就是如何让某种东西成为人们**想要的**。如果想要某种东西，你的客户就必须知道它已经存在并且觉得它正是自己想要的。它不会从天而降，所以我们要使用正确的位置和诱饵。在创业领域，这是由口口相传、市场推广、销售和品牌化组成的。刚开始的时候，营销策略主要是由创始人自己去销售。随着公司的发展，口口相传、市场推广和品牌化对于公司规模的扩大都是必不可少的。具体使用哪一种策略，取决于我们所做的产品类型。表 4-1 列举了现实中的几个例子。

表4-1：不同产品的主要营销渠道

客户数	产品类型	价格范围	示例公司	营销策略
10 亿	实物	1~10 美元	可口可乐	市场推广（广告宣传）
10 亿	广告	1~10 美元	Facebook	口口相传（网络效应 + 病毒式传播）
1 亿	实物	10~100 美元	强生	市场推广（广告宣传）
1 亿	广告	10~100 美元	TripAdvisor	市场推广（用户生成内容 +SEO）
1000 万	视频游戏	100~1000 美元	暴雪	市场推广（广告宣传）+ 口口相传（病毒式传播）
100 万	SaaS	1000~1 万美元	sendwithus	市场推广（集客式营销）
10 万	企业支持	1 万 ~10 万美元	MongoDB	销售（内部销售）
1 万	数据分析	10 万美元以上	Cloudera	销售（外部销售）

综上所述，如果你有好的点子、设计、数据和营销，就可能做出人们想要的东西。

> 点子不是设计，
> 设计不是原型，
> 原型不是程序，
> 程序不是产品，
> 产品不是企业，
> 企业不是利润，
> 利润不是退出，
> 退出也不是快乐。
>
> —— Mike Sellers，连续创业家

第二部分

技术

第 5 章

技术栈的选择

5.1 关于技术栈的考虑

我们应该使用哪种编程语言、哪种 Web 框架呢？如何储存数据？在创业公司中应该使用什么技术栈？

技术栈就是工具，它是实现产品的手段，不是产品的终结，也不是产品本身。不要因为某项技术听起来很酷或者很有趣就选择它，我们选择一种技术是因为它可以为我们所用。为此，应该在把选择技术栈的**黄金法则**记在心中。

> 好的技术栈的扩展要快于需要进行的维护。

我们的目标就是要能够对技术栈进行扩展——以支持更多用户、更多流量、更多数据和更多代码——通过投入资金和硬件，而不是投入人力，来解决问题。如果用户库翻倍，只需要多购买几台服务器，一切就可以正常运转，那就是良好的状态。换句话说，如果必须把团队人数翻一倍才能应付得了，可能就需要做些改变了。记住，创业与人是密不可分的，即便技术是本章关注的重点，但技术最重要的还是它能够为使用它的人带来什么作用。

谈到技术的作用，WhatsApp 团队就是一个很好的例子。该团队基于 Erlang 搭建了一个技术栈，可以支持每秒 7000 万条 Erlang 消息、4500 万用户，每天 500 亿条消息，每年 7.2 万亿条消息[1]，而完成这一切的团队只有 32 个工程师。

当然，这里讲 WhatsApp 的故事并不是说所有人都应该用 Erlang。各种成功的创业公司都会基于他们所有可能想得到的技术，去实现他们的产品。这一章会帮助你解决“创业中可以使用什么样的技术栈”这一问题。首先，我会描绘如何选择最初的技术栈，如何随着时间推移使它逐步进化；接着，将把实现内部技术方案、购买商业软件和使用开源软件之间的权衡利弊呈现给大家；最后，我会深入谈谈创业公司最常遇到的 3 个技术决定

注 1：作为参考，全球所有电信机构之间每年发送的 SMS 消息数量是 7.5 万亿条。

的一些细节，即编程语言、服务器端框架和数据库。第 8 章还将讨论技术栈其他方面的内容，包括如何构建、部署和监控代码。

5.2 技术栈的进化

本书的主题之一就是伟大的公司是进化的结果，而不是天才的设计，同样的道理也适用于这些公司的技术栈。尽管我们喜欢认为技术是严谨计划的结果，就像城市会根据蓝图进行网格式的详细布局规划。但现实情况是，大部分技术栈是自然发展的结果，看起来更像是荒草丛生，满是为了生存所需而到处乱长的树根和分枝。这是由于我们几乎不可能预测未来将面对什么样的技术挑战，而且实现自己不需要的东西也是一种浪费。因此，唯一可以做的就是先从小型、简单的技术栈开始，并创建一个在必要的时候能够改造技术栈的进程，使之适应来自环境中的新压力，比如需要处理增长的流量、新来的员工和新的功能。换句话说，我们更应该关注如何构建可以不断进化的技术栈，而不要太过关注现在什么是“最佳的”技术栈。

事实上，也不存在所谓“最佳的”技术栈。如果选择技术时没有考虑产品类型、团队和公司文化，就好比在买房子、做预算或者在弄清和谁一起住之前，就先决定了要买什么家具。所以，情景是至关重要的。

例如，像 Google 这样的公司必须保持它的技术栈能够支持惊人的规模扩展。Google 工程师必须很小心，保证他们的网页爬虫不会把小网站弄得不堪重负；他们还必须为项目制订周密的计划，因为他们所处理的都是以 PB 为单位的数据；他们还得仔细考虑如何测试排名算法，因为亿万用户正在使用搜索结果作为日常生活的指引。另一方面，AdMob 这样的创业公司在 2007 年的时候，对技术栈却有着完全不同的需求。

> 当我（离开 Google）到了 AdMob 后，我心中的第一个冲动就是“噢，垃圾垃圾垃圾”。当然我没有说出口，因为作为领导者，学会的更重要的一课就是在开始说之前先闭上嘴巴去听。但是我脑子里想的是：“在 Google 我们是不会这样做的，在 Google 我们是不会这样做的，在 Google 我们是不会这样做的。”这句话在脑海里闪过了好多次之后，我才意识到这并不是在 Google，我要解决的是不同的问题，面对的是不同的技术文化——这是很好的事情，真的很好。
>
> 在 AdMob，我们处于市场快速变化的环境下，只有先行者才能从中受益，时间几乎是以月来计算的。要取得成功，就必须做到不可思议的敏捷。所以整个系统的设计都以敏捷为目的：我们招聘的都是一些酷爱刺激的人；我们要确保你在第一个星期就能把代码写出来并放到产品中；如果无法做到，我们在第二周开头就会和你进行一次很严肃的交谈，判断这是否真的是适合你的环境。
>
> 我们鼓励公司的人进行在其他人看来也许是很疯狂的冒险。公司中的每一个工程师都对数千台机器有 root 访问权限。对于工程师来说，可以随时从运转的机器中找出一台来，部署新的广告服务程序上去，这是再美好不过的事。哪怕部署出现了问题也完全不怕（即使是突如其来的中断），因为我们对整个系统进行了精心设计，对软件中预判的关键位置进行了监控。如果有地方出现了严重的问题，会有一个狙击手射中坏蛋，并发通知让别人过来收拾烂摊子。

> 我们把这样的方法用在核心数据库，用在生产环境的广告服务进程、人员招聘、软件开发过程，以及安全网络的建立上。我们营造了这样一个鼓励冒险的环境，从而获得了这些疯狂的先行者给我们带来的优势。这样的方法甚至已经融入团队结构中：我们尝试尽可能简化团队结构，希望能够把事情完成后就转到下一件事，不要有什么编码人员、经理等人为设计出来的关系在其中。因为，也是再次要强调的，敏捷才是重要的。
>
> 我记得在 2008 年 6 月，乔布斯在 WWDC 上宣布 iPhone 即将拥有一个应用商店。我想所有人都知道了这个消息，因为 Apple 在那一年的 3 月已经宣布了公测计划，但我们并没有被纳入其中，所以我们完全不知道任何细节。我非常想看主题演讲的直播博客，所以那天我都待在家里，以免演讲开始时发生堵车。我记得乔布斯在直播中揭开了应用商店运行的所有细节，并介绍了开发人员可以做些什么。在 WWDC 结束的时候，我的电话立即响起。电话是 Omar 打来的，他是公司的 CEO。他说："你刚刚看了乔布斯的主题演讲了吗？"我说："是的。"他又说："我们必须搭上这趟车。我们已经为这个应用商店做了广告 SDK，必须在应用商店推出前把它完成。"
>
> 好吧，应用商店会在**六个星期后**推出，现在已经是 6 月了，那么应用商店应该会在 7 月中旬推出。因为已经具备了适应敏捷的能力，到那天结束时，我们已经组建起了一个完整的团队，共有六个人。我们说："我们需要把这个东西做出来，它实在太棒了，我们要把它做出来。我们要成为第一家有应用内广告 SDK 的公司，我们要立即做出来。你们只有六个星期的时间。"
>
> 团队并没有慌乱。没有人会因为从手头的工作中抽离出来而慌乱，他们被抽离出来也没有影响到其他任何事情的运作。他们都尽情投入，沉迷于把东西做出来。我们在六周内把它做好并发布出来，并随着应用商店的推出同时运行并发压力测试。6 个月后，它已经成为我们最大的业务。
>
> ——Kevin Scott，LinkedIn 高级副总裁、Admob 副总裁、Google 主管

由于不了解你所处的具体环境，我在本章无法为你做出特定的技术推荐。作为替代，我的目标是让你了解做决定时需要考虑的各种概念，让你意识到其中的权衡取舍，向你解释其他公司在过去是怎么做出那些决定的。

我们先从需要在很早就做出的一个决定开始：如何为创业公司选择初始的技术栈？我可以只用一句话来回答：熟悉什么就用什么。这句话源于我为这本书所做的访谈，我可以告诉你，每一家创业公司选择的只不过是创始团队最精通的技术。LinkedIn 是用 Java 实现的，因为创始团队了解 Java；GitHub 的创始人全部都是 Ruby 开发者，所以他们也是用 Ruby 去实现网站的；Twitter 主要用的是 Rails，因为他们的早期员工中有许多人熟悉 Rails。Foursquare 开始时使用 PHP，因为这是联合创始人 Dennis Growley 所了解的；Pinterest 使用的是 Python，因为创始团队对 Python 比较熟悉。

学习一门新的技术、享受它承诺的所有理论上的好处可能很好玩，但在创业早期，我们的目标是认识到用户需要什么，其他任何花时间的事情都是浪费。在早期，产品的用户和代码都不多，所以可扩展性并不是太大的挑战，我们要做的就是尽可能快地进行迭代（阅读 2.2.1 节了解更多信息）。如果你是 Java 专家，就用 Java；如果你喜欢 Ruby on Rails，

就用 Ruby on Rails；如果你已经用了 MySQL 好多年，就用 MySQL。一个能够立即实施的出色的技术栈，好过一个下周才能运用完美的技术栈[2]。

当然，如果你的创业公司足以成功地生存到“下一周”，也许你可以让技术栈有所进化，使之满足新的需求。例如，Twitter 开始时用的都是 Ruby on Rails，但发展起来之后，就不得不迁移到 Scala 和 JVM 上；HubSpot 从 .NET 和 SQLServer 迁移到 JVM 和 MySQL、Hadoop 及 HBase；Coursera 现在从 PHP 迁移到了 Scala；LinkedIn 在它的发展历史中已经尝试了十多种技术，包括 Java Servlets、Groovy on Rails、JRuby on Sinatra、Java 和 SpringMVC、JavaScript 和 Node.js，以及 Scala 和 Play Framework。

> 最初选择的技术并不是关键，最初的决定在最后看来一定是错的，唯一的问题是它会错多久。关键在于，你要在遇到拐点时有壮士断腕的勇气，而不是为了存活而一层一层地给它贴上创可贴。
>
> 遵循这样的准则极其重要。从目前来看，这比紧跟潮流、为了做出初始的最佳技术决定而进行无穷无尽的设计分析要更为重要。我们应该做的是，确保把自己和环境锻造成能够适应各种变化，能够知道什么时候是重建的合适时机。
>
> ——Kevin Scott，LinkedIn 高级副总裁、Admob 副总裁、Google 主管

了解何时改变你的技术栈从根本上说是可扩展性问题。当你违背了技术栈的黄金法则——当你发现人数的扩展快于技术的扩展——就是时候重新进行评估了。如果新功能的实现比你预料的时间还要长，每次发布新版本影响到的功能比新增功能还要多，也许就是时候去改变了。我们有时必须要置换技术栈中较大的部分，比如迁移到不同的数据库，但是要警惕使用停止一切、彻底重写的做法。

暂停所有的发展、在全新的技术栈上重写代码要冒巨大的风险。这种情况被称作“所有软件公司都可能犯的单一的、最糟糕的战略性错误”和“创业自杀”。如果你抛开旧代码，就等于抛开了多年的学习和修复的 bug。在重写的时候，你终将重复面临许多相同的错误，还要加上许多新的错误。你会意识到，用全新的技术去重写代码只是问题的一小方面，你的大部分时间将花费在重新培训团队成员用新的方法去做事情，说服他们新方法比老方法好，还要更新文档，处理数据迁移的问题，将该技术整合到构建和部署系统中，设置监控，解决调试新技术的方法。代码的重写就是侯世达定律（Hofstadter’s Law）的最佳例子：你做事所花费的时间总是比你预期的长，即便你已经考虑了侯世达定律。与此同时，你的产品正深陷泥潭，你的竞争者正在超越你。

那么，要如何让技术栈进化且不会扼杀你的创业呢？答案就是**渐进主义**。其思路就是把任务分解成小的、孤立的步骤，每一个任务都有它自身的价值。但并不是所有“小步骤”都是生而平等的，所以要警惕**错误的渐进主义**。

> 错误的渐进主义就是把大的改变分解为一系列小的步骤，但是这些步骤本身并不能产生任何价值……幸好，有一个非常简单的测试可以判断你是否陷入了错误的渐进主义中：如果每一次增量之后，都有一位“重要人物”让你的团队在那一刻立即退出项目，你做得这些事情是否还有价值？这就是黄金标准。
>
> ——Dan Milstein，Hut 8 Labs 联合创始人

注 2：这句话是对巴顿将军的致敬。

简而言之，即便你必须对技术栈做出重大调整，最佳的方法还是对现有的东西进行渐进式地进化，而不是抛开一切，尝试从头找出替代方案（即进化胜过天才的设计）。某种程度上，这就有点像让车子在行驶过程中换轮子。但创业领域是没有停车道的，如果你靠边提车，必死无疑。

举个例子，2011 年前后，LinkedIn 进入了一个高速发展期，网站流量和雇员人数都大幅增长，底层架构已经不堪重压。我是服务基础框架团队的一员，我们知道必须做些重大改变，让技术栈能够扩展以适应快速增长的需求。其他团队致力于对代码交付进行大量必要调整（阅读第 8 章了解更多信息），而我们的团队则致力于改进最早编写的代码。最终，我们启动了一个项目，将 LinkedIn 迁移到了 Play Framework 上。表 5-1 展示了我们为完成这一决定所采取的增量式步骤，以及是如何执行实际迁移的，其中包括如果项目成功的话，在项目的每一个阶段都将发生什么；如果项目被取消的话，为什么还是值得这么做。

表5-1：增量式地将LinkedIn迁移到LLEPlay Framework上

阶段	如果项目成功	如果项目被取消	实际结果
阶段 1：与开发团队交谈，找出他们最大的痛点，找到一个“早期采纳者式的团队”，乐意尝试把新技术作为解决方案	能找出如何划分基础框架工作的优先级	能知道是什么伤害了团队，在获得资源继续推进之前先把工作暂存起来	发现 Web 框架的生产力和性能是造成巨大痛苦的原因，决定尝试把 Play Framework 作为解决手段
阶段 2：实现最少的集成点，以将 Play 用在 LinkedIn 上	得以实现基本的集成代码，在 LinkedIn 上支持 Play 应用	我们所学到的经验教训和部分集成代码对于未来引入任何框架都是有帮助的	把 Play 纳入监控、部署和配置工具中
阶段 3：和早期采纳者团队一起，基于 Play 重写其中一个服务	至少有一个团队可以从提升的生产力和性能中受益	至少有一个团队可以从提升的生产力和性能中受益	用 Play 重写了 LindedIn 的 Polls 后端
阶段 4：回到阶段 1	发现有更多团队对迁移到 Play 感兴趣，实现了他们所需要的新的集成点	发现其他团队存在不一样的痛点，把关注点转移到解决那些痛点上	把 80 多个服务迁移到 Play，包括主页、工作、招聘和 Pulse，迁移工作仍在进行中

也许你已经注意到，这就是迭代式产品开发的过程，与本书第一部分的讨论类似。这样的过程能够增量式地将网站的各个部分迁移到 Play，渐进式地获得性能和生产效率上的提升，完全不用冒重写重大代码而必须暂停一切的风险。每个步骤本身也都是有价值的，所以不管我们在什么时刻必须要停止项目了，所做的事情仍然是有价值的。

Play 项目成功的原因之一就是我们不需要从头开始开发全新的 Web 框架。我们并没有自己去实现大规模的基础框架（这很难去增量式地实现），而是使用了开源产品，并且签订了商业支持合同。

5.3　内部实现、购买商业产品，还是使用开源产品

对于技术栈的每一部分，我们都要判断应该是内部去实现、购买商业产品，还是使用开源项目。

5.3.1　内部实现

我们可以对内部实现的项目实现完全控制。可以拥有代码和数据，也可以根据自己的需要定制项目，根据自己的意愿发布新的功能，自己决定产品在未来如何发展。内部实现是一次性定制产品（比如网站的用户界面），以及所有独特之处（比如 Google 的网页排名算法）唯一的选择。但是，如果涉及的是可重用的库或者基础框架，实现专有的软件也会带来巨大的成本：开发时间。

大部分开发人员只会考虑编写项目初始版本所花费的时间——而且通常都会远远低估这一时间——但这还只占总成本的极小一部分。这些开发人员还必须去长期维护项目，对它进行改进以满足新的需求，修复 bug，创建文档。只要出现问题，同样的这群开发人员，只要他们还在公司里，就要负责回答所有的问题并提供 7×24 小时的支持。Stack Overflow 对这些专用代码也无能为力，也没有社区可以贡献插件和扩展，你也雇不到一个已经是该项目专家的人。事实上，大多数开发人员都不喜欢学习专用的系统，因为他们无法把这些知识用在职业生涯的其他地方。

5.3.2　购买商业产品

商业产品其实就是用金钱去换取开发人员的时间，由外部厂商负责编写所有代码、修复 bug、创建文档。既然有一整家公司专注于这样一件产品，他们可以投入的人力肯定比你的创业公司要多得多。有些厂商也提供支持合同，我们可以付钱去定制产品，优先要求修复 bug，获得每天 24 小时的帮助。有些商业产品甚至还有社区支持，产品社区可以很好地提供产品情况的证明，还有已经熟悉如何使用软件、插件和扩展的开发人员，可以通过 Stack Overflow、邮件列表、博客文章和演讲等形式为你提供帮助。由声誉良好的厂商做出来的、已经被十来家公司成功使用的产品，与你从内部开始做起、完全未经检验的项目相比，会是更加安全的选择。大部分创业公司对商业软件的利用程度都比较高，特别是 SaaS 提供了 Slack（团队沟通平台）、PagerDuty（监控和报警工具）、Amazon EC2（云主机）、Zenefits（线上人力资源软件）、Salesforce（客户关系管理）等产品以及其他一些服务。

使用商业软件自身也会有成本，部分就体现在厂商发给你的账单上，还有一些则没有那么明显。例如，厂商掌握着代码，所以你是看不见它的，你也无法判断它的质量，不清楚它是否是安全的；需要调试问题的时候，你无法参考代码，也无法控制它在未来能如何进化。如果你无法获得 SaaS 类产品的代码，以后要将数据迁移到不同的技术平台上是非常困难的（这就是所谓的厂商锁定）。

> 一个残忍的事实是：当你的关键业务过程运行在内部情况不清楚（更别提修改）的不透明代码上时，你就失去了对业务的控制。你对供应商的需要超过了供应商对你的需要——因为这一巨大的不平衡，你只能付钱、付钱、再付钱。
>
> ——Eric S. Raymond，《大教堂与集市》

简而言之，每当使用商业产品时，就是将公司的一部分投注在无法控制的第三方身上。如果这家厂商在几个月后关门歇业了怎么办？或者被竞争者收购了呢？这些都是巨大风险，所以一定要警惕在业务的关键部分使用未经检验的厂商提供的产品，比如数据存储。

5.3.3 使用开源产品

开源产品可以提供很多商业产品具备的好处，但是又不需要面对那么多风险。对于开源产品，有社区中的开发人员负责编写代码、修复 Bug、建立文档、开发插件和扩展。这样的社区是我们获得帮助的强大资源：我们的问题可以在 Stack Overflow 和邮件列表上得到回答，我们可以从博客文章和演讲中学到最佳实践，可以雇到已经非常精通这些产品的开发人员。事实上，大多数开发人员都喜欢在开源产品上工作，因为他们可以在后续的职业中重用这些知识，他们为开源项目所做的贡献都可以放在公开的简历中（阅读 12.2.3 节了解更多信息）。对于一些流行的项目，开源社区比任何单一的公司都要大得多。例如截至 2014 年，已经有 702 人为 Diango 提交了代码，有 2469 人为 Ruby on Rails 提交了代码，每一种框架都有数以千计的插件。如果你的创业公司正在考虑编写自己的 Web 框架，你觉得会有多少人可以把时间精力用在它上面呢？

开源项目的不利之处在于它主要都是基于志愿者完成的，用这种方式让来自许多公司的大量开发人员一起工作是很有效的（阅读 9.5.1 节了解更多信息），但如果需要任何保证，比如修复特定的 bug、在特定的截止期限之前发布新的版本，甚至保证项目在任何情况下都会继续开发下去，开源项目在这些方面的效果就不是太好。有时候，开源项目的维护者会将其完全放弃；有时候，维护者甚至会删除项目（例如被 Apple 收购之后，FoundationDB 删除了 GitHub 上所有的文件）；有时候，项目的社区会分裂成不同的方向（由于对 Joyent 支配 Node.js 的不满，导致出现了名叫 io.js 的分支）；有时候，开源项目并不像人们认为的那样“开放”，经常存在许可（例如大多数商业公司都要避开 GPL 许可的软件）、商标（例如 Joyent 拥有 Node.js 的商标，所以 Node.js 的分支就不能在名称中使用“Node”）以及版权上（例如现在有 Java 的开源实现，而 Oracle 声称对 Java API 拥有版权，并控诉 Google 在 Android 中抄袭了这些 API）的混乱。

就像商业软件一样，使用开源软件意味着要让公司承担完全无法控制的、来自第三方的风险。和商业软件不同的是，我们可以获得开源软件的源代码，所以在一定程度上降低了这样的风险。如果掌握了代码，就可以贡献补丁和插件；如果发布周期太慢，可以创建定制的版本；如果项目开始朝着错误的方向发展，可以为项目创建分支，或者在必要的情况下迁移到完全不同的项目（没有厂商锁定）。而且既然其他人也都在关注开源代码，表明开源项目比专有项目质量更高，也更安全。我们可以通过阅读代码、考量有多少公司正在使用它、维护者的声誉如何、有多少关注者和分支、网上有多少可用的资源，更好地衡量开源项目的质量。如果需要帮助，也有专门为开源项目提供商业支持的公司，比如 RedHat、Typesafe、Joyent、Cloudera 和 Hortonworks。

5.3.4 永远不要自己实现的技术

如果某些技术自己实现起来非常复杂、很容易出错、要花大量时间，但开源和商业领域已经有很好的方案，那么作为创业公司，就永远不要自己去实现这些技术。下面列出了

部分这样的技术。

- 安全：加密、密码存储、信用卡存储。
- Web 技术：HTTP 服务器、服务器端和客户端框架。
- 数据系统：数据库、NoSQL 存储、缓存、消息队列。
- 软件分发：版本控制、构建系统、自动化部署。
- 计算机科学：基本数据结果（映射、列表、集）、排序算法。
- 处理通用数据格式的库：XML、HTML、CSV、JSON、URLs。
- 实用库：日期 / 时间操作、字符串操作、日志记录。
- 操作系统。
- 编程语言。

如果你要从头开始实现上述系统中的任何一个，只能有两个原因：一是以学习为目的的个人项目，二是你的创业公司对其中的某项技术有极其独特的需求。第二种情况是很少见的。如果你的业务就是销售数据库，或者你的处理规模是其他任何公司都无法比拟的，实现自己的技术才是有意义的；否则，请使用现成的解决方案。

> Google 就是一个很明显的“非自主发明不可”的栈，它所有的一切都是内部编写出来的。在 Google 的时候，我想可能除了 gcc 之外，我没用过任何一种开源工具或库。部分原因是 Google 领先行业内其他所有公司 5 年或 5 年以上。Google 所做的东西，就是像 MapReduce（使用无数低成本的商业硬件去运行分布式系统）这样的产品，他们基本上都是在发明和普及很多这样的产品。这些产品现在全都成了行业标准，但是大部分在 Google 之前并不存在。我觉得 Goolge 就是因为比其他公司超前了许多，所以不得不去实现，这样的境况也许又成为了一种自我增强，因为我们已经形成也适应了非自主发明不可的文化。
>
> ——Brain Larson，Google 和 Twitter 的软件工程师

5.3.5 结语

对创业公司来说，使用开源产品通常是最好的选择，其次就是使用商业产品。内部实现基础框架应该被看作是最后的选择，只有在没有其他选择的时候才可以用。人们可能很难记住这一点，因为许多开发人员一有机会去实现复杂的基础框架都会变得兴奋起来，所以很快就声称没有现成的技术可以满足他们的需要。但考虑到有超过一千万的开源资源可供选择，而且创业公司中开发人员的时间是最稀缺和昂贵的资源，在现成的解决方案唾手可得的情况下，我们不可能承担重新发明轮子的时间成本。利用图 5-1 中的流程图，我们可以在内部实现、购买商业产品和使用开源产品之间做出选择。

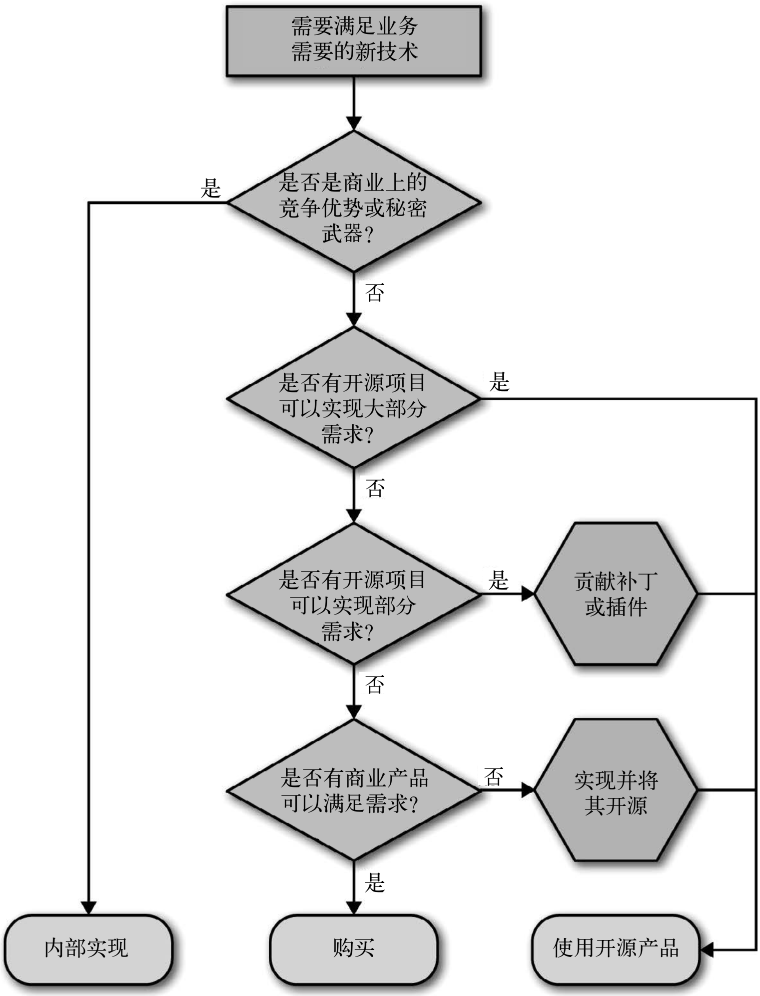


图 5-1：内部实现、购买商业产品或使用开源产品

5.4 选择编程语言

选择编程语言通常是要做的第一个技术上的决定，同时也是对其他决定最有影响的一个。“熟悉什么就用什么”的原则意味着大多数创业公司开始所使用的都是创始人最了解的语言。然而，随着创业公司的发展和进化，通常都要引入其他语言。例如，Twitter 开始时用的是 Ruby，但是在最近几年，他们已经把许多服务迁移到 Scala。但为什么是 Scala？为什么不是 Python、Java、Haskell 或者其他语言？要回答这个问题，需要弄清楚每种编程语言之间的关键差别，包括编程范式、适用问题、性能、适用文化和生产效率。

5.4.1 编程范式

每一种编程语言对于如何解决问题都有不同的哲学。我们可以把编程语言的范式当作是该语言的词汇和语法，决定了你如何说和可以怎么说。目前很少有确切的证据表明某种范式优于其他范式[3]，但某些范式相比其他范式可以更方便地表达一些想法。接下来的几节将讨论少数几种最主流的范式之间的利弊，包括面向对象编程、函数式编程、静态类型和自动内存管理。

1. 面向对象编程

面向对象编程（object-oriented programming，OOP）尝试将世界万物用对象（即封装了数据和行为的数据结构）来建模。OOP 在近 20 年来一直都占据着编程范式中的统治地位，世界上最流行的一些编程语言都使用了这一范式，包括 C++、C#、Java、JavaScript、Ruby 和 Python。

面向对象编程之所以这么流行，部分因为对象和方法通常恰好能与现实中的名词和动词对应起来。我们可以凭直觉想出一个带有 `move` 方法的 `Car` 类，`move` 方法知道如何更新 `Car` 的内部状态。OOP 鼓励**信息隐藏**的方式有助于减少耦合。所谓信息隐藏，就是一个对象不会让其他对象访问其内部实现细节，而这些细节是很可能发生变化的。相反，程序的其他部分必须通过对象的公共方法与之进行交互，这些公共方法是更加稳定的接口。

OOP 存在两个主要的问题。首先，对于"面向对象"真正意味着什么或者怎么做才是正确的并没有共识。每一种 OOP 语言和每个程序员的做法都是不一样的[4]。其次，大部分 OOP 语言都鼓励使用多态和副作用（side effects，阅读第 6 章了解更多信息），使得推导、维护和测试代码都变得更加困难，特别是在并发环境下。

2. 函数式编程

函数式编程语言尝试将世间万物用函数的求值来建模。和 OOP 编程不同，它着重限制了可变数据和副作用的使用。函数式编程关注的是将单纯的函数组合起来，使用一种更加声明式的编程风格（即描述想要实现什么，而不是如何去实现）去构建复杂的代码。这使得函数式编程更容易推导、维护和进行代码测试（6.7 节将更深入地讨论这一主题）。流行的函数式编程语言包括 Haskell、Lisp 系列（即 Scheme 和 Clojure）以及 Scala（这是 OOP 和函数式编程的混合）。

为什么函数式编程没有像 OOP 那样流行呢？有两个主要原因。一是因为函数式编程的学习曲线比较陡峭。范畴论、monad、monoid、applicative 和 functor 这些概念比"做事情的对象""猫由动物扩展而来"这些概念更难入门，大众理解起函数式编程的数学根源相对困难；二是因为从设计的角度看，函数式编程并不接地气——它抛弃了副作用和状态，但大多数程序的存在仅仅只是为了维护状态以及能够与外界进行有趣的互动。函数式编程也抛弃了底层硬件架构。例如，它使用递归来代替循环，使用不变数据代替可变数据，使用垃圾回收代替手动内存管理，使用惰性求值（lazy evaluation）代替迫切求值

注 3：只有少量证据表明，静态类型和函数式编程在生产效率上有适度的提升，但是这些范式之间的差异又容易被程序员之间能力上的差异极大地掩盖。

注 4："我可以告诉你，我创造面向对象这个术语时脑子里并没有 C++""Java 是自 MS-DOS 以来使用计算机时最难受的事情"，面向对象编程的发明者之一 Alan Kay 是这么评价两种最流行的 OOP 语言的。

（eager evaluation），这些使得函数式代码的性能更加难以预测。有许多方法可以减轻甚至消除函数式编程在性能上的损失，比如使用持久化的数据结构和尾部调用优化（tail call optimization），但这些方法通常都会增加程序员的负担。

3. 静态类型

在编程中，每个数据都有它的类型，这决定了数据在内存中是如何存储的，它可以有什么值，我们可以对它执行什么操作。例如，在一些语言里，如果一个值是 `int`，意味它在栈中是以 32 位有符号二进制补码整数的形式存储的，可能的值从 -2^{31} 到 $2^{31}-1$，有效的操作是加、减、乘、除和求余。**动态类型语言**只会在运行时检查类型，如果尝试访问的数据索引超出了边界就会抛出错误。**静态类型语言**能够在编译期捕捉确定的类型错误，如果尝试把一个 `String` 赋给 `int`，编译的时候就会失败。

静态类型系统就像编译器强制执行的一套自动化测试，随时保证类型是正确的（阅读 7.2.1 节了解更多信息）。但即便我们使用的是静态类型的语言，也可以（并且应该）编写自动化测试，类型系统可以自动捕捉到大部分 bug，从而节省大量时间。静态类型也为阅读代码的人提供了许多有用信息，其中包括开发人员，他们可以把类型签名作为一种文档；还有 IDE，可以使用类型签名更容易地实现代码导航、重构和自动编译等功能；而编译器则可以使用类型签名对代码进行优化。

但静态类型也不是万灵药。静态类型的代码必须被编译，会花费时间，降低迭代速度。而且不管我们花多长时间编译，只有一部分代码的"正确性"会得到静态检查。随着类型系统变得更加强大，这个范围也在扩大，但是通常都是以类型系统复杂度的指数式增长为代价的。在学习语言时，我们还不得不学习它的全部内容——泛型 (generics)、协变（covariance）、逆变（contravariance）、实存类型（existential types）、唯一类型（uniqueness types）、联合类型（union types）、依赖类型（dependent types）、自递归类型（self-recursive types）、类型类（type classes）、类型限界（type bounds）、高阶类型（higher kinded types）、虚位类型（phantom types）、结构类型（structural types），而有时使用一门语言的开销比它能给我们带来的好处更重要。对某些类型的问题尤为如此，比如领域特定语言（domain-specific languages，DSL）和元编程（metaprogramming），它们在灵活性和可表达性上都超过了大部分类型系统。

4. 自动内存管理

低级的系统编程语言，比如 C 和 C++，需要程序员手动管理内存的分配和释放。大多数高级编程语言，比如 Java、Ruby 和 Python，都支持自动内存管理，可以让程序员关注于所解决的实际问题，而不是计算机底层的内存架构。这样既提升了生产效率，又能防止很大一部分 bug 出现，比如忘记释放不再使用的内容（内存泄漏），或者在错误的时间释放内存（迷途指针和双重释放的 bug）。

不幸的是，自动内存管理也是有代价的。自动释放内存最常见的方式就是**垃圾回收**（garbage collection，GC），即定期运行**收集器**去扫描所有已分配的内存，对不再使用的内存进行回收。问题就在于收集器的运行会消耗 CPU 和内存资源，虽然通过调优可以降低这一开销，但对于内存密集型的程序来说开销仍然过多，比如高性能的内存缓存。许多垃圾回收算法还要求在收集期间暂停整个程序，就意味着提供垃圾回收机制的语言不是实时应用程序的好选择。所谓实时应用程序，就是必须在非常短的时间内随时响应的程序（也就是说，比典型的垃圾回收暂停时间更短的周期）。

5.4.2 适用问题

理论上来说，所有现代编程语言都是图灵完备的，所以它们都是等价的。在实践中，一些编程语言解决起某些类型的问题，比用其他语言更方便。例如，有强元编程能力的语言，比如 Clojure 和 Ruby，可以方便地定义出自定义的 DSL；Erlang 在实现容错的分布式系统方面特别有效；汇编和 C 通常都是底层、实时或嵌入式系统的唯一选择。

一种语言的社区活跃程度对适用问题也有显著的影响。例如，C++ 和 Python 拥有大量的计算机视觉库；Matlab、Mathematica 和 R 语言拥有全面的数学、绘图和统计库；PHP、Ruby、Python、JavaScript 和 Java 有着庞大的生态系统，能提供实现 Web 应用程序的各种库及框架。对于某些问题领域，选择合适的语言可以带来生产效率上的巨大提升，因为许多代码已经为你编写好了。

5.4.3 性能

编程语言对大多数公司来说通常都不会是瓶颈所在（阅读第 7 章了解更多信息）。然而，在某些情况下，特别是出现了足够的负载之后，语言就变得很重要了。垃圾回收和并发性是编程语言最常见的两个性能瓶颈。

5.4.1 节已经讨论过，垃圾回收会消耗 CPU 和内存，并且会暂停程序的执行。有一些垃圾回收算法比起其他算法更加成熟、可调整性更高。例如，JVM 以拥有更好的垃圾收集器著称，而 Ruby VM 的垃圾收集器也因众多性能问题而为人所知[5]。然而，这两种语言在性能方面都无法与没有垃圾回收的语言相比。如果你的应用程序不能容忍任何的 GC 暂停或者 CPU、内存的开销，也许可使用手动内存管理的语言，比如 C 或 C++。

对于并发性，最重要的因素就是一门编程语言支持什么样的并发结构，以及如何处理 I/O。例如，Ruby 支持线程，但它有**全局解释器锁**（Global Interpreter Lock，GIL），这意味着一次只能执行一个线程。此外，大多数主流的 Ruby 库执行的是同步 I/O，在等待磁盘读取或网络调用返回的时候会阻塞线程。这导致了 Ruby 并不是处理大量并发的高效语言。目前也有一些解决方案，比如运行多个 Ruby 进程（即每个 CPU 内核运行一个）、使用非阻塞的库（例如 EventMachine），或者使用不同的 VM（例如 JRuby），但是这些都需要一定的利弊权衡和开销。

这也是 Twitter 离开 Ruby 转向 JVM 的原因之一。JVM 完全支持多线程，没有全局解释器锁，也完全支持非阻塞 I/O 和多种多样的并发结构，包括线程和锁、Futures、Actor 和软件事务内存（Software Transactional Memory）。从 Ruby 到 Scala 的迁移帮助 Twitter 将搜索延迟减缓至 1/3，CPU 使用缩减了一半。

5.4.4 生产效率

虽然编程语言的性能很重要，但对于大多数创业公司来说，程序员的性能才是更大的瓶颈。我们要寻找一门能够以最少时间去完成最多工作的语言。生产效率涉及两个主要方面：有多少现有的代码可以重用，以及你能够创建新代码的速度有多快。

注 5：Ruby 2.1 对垃圾回收进行了很好的改进，但仍然有许多遗留问题。

现有代码的数量取决于这门语言的流行程度以及社区的规模。流行的语言拥有更多的学习资源，我们也可以雇到更多已经熟悉该门语言的人，可以使用更多的开源库。成熟语言的生产力工具也是一个生态系统，比如 IDE、分析器、静态分析工具和构建系统。我们可以重用的代码越多，需要自己编写和维护的代码就越少。

创建新代码的速度取决于三个因素。第一个因素是经验，你对一门语言的经验越丰富，生产效率就越高，所以要寻找你和团队已经熟悉并且有丰富文档且易于学习的语言。第二个因素是反馈循环，就是代码修改后需要多长时间才能看到效果。如果必须等待几分钟才能完成代码编译和部署，与只需要等待几秒钟让页面刷新或让脚本返回相比，前者的生产效率更低。我们要寻找支持热重载（hot reload）、具有交互式编码环境（比如“读取－求值－输出”循环，REPL）、快速编译、快速自动化测试的语言。第三个因素是语言的表达能力，即对于任何给定的想法，需要多少行代码才能实现。需要编写和维护的代码行数越多，面对的 bug 就越多，前进的速度也越慢（阅读 7.2.2 节了解更多内容）。一般来说，我们应该在满足其他需求的前提下，尽可能挑选最高级和简洁的语言。

5.4.5 结语

> 当你选择一门语言时，面对的不仅仅是技术上的权衡取舍，而是一个社区。就像选择一间酒吧一样，没错，你去酒吧是为了品尝美酒，但那还不是最重要的——酒吧更是人们休闲和聊天的地方。这和选择计算机语言的道理是一样的，一门语言随着时间的推移会建立起社区，不仅仅是人，还包括软件方面的产物：工具、库等。这就是为什么有一些语言理论上比其他语言要出色，实际却不如其他语言的原因之一——它们还没有建立起健全的社区。
>
> ——Joschua Bloch，Sun 公司分布式工程师、Google 首席 Java 架构师

尽管有数以百计的编程语言可供选择，但足够成熟并且有足够社区支持、可以作为创业公司选项的语言屈指可数。下面是 2015 年的一份清单，是根据编程语言流行指数（TIOBE、LangPop 和 RedMonk）、Stack Overflow 开发人员调查和我自己的经验得出的，按字母排序：

- C 系列（C、C++、C#）
- Go
- Groovy
- Haskell
- Java
- JavaScript
- Lisp 系列（即 Clojure 或 Scheme）
- Perl
- PHP
- Python
- Ruby
- Scala

我们可以应用三个过滤条件快速缩短这一列表：适用问题、编程范式和性能需求。例如，一家做计算机视觉和机器学习系统的创业公司，应该根据适用问题把这份列表限制为三门语言：C++、Java 和 Python。如果该创业公司更偏爱静态类型，就可以把 Python 从列表中剔除。最后，如果他们实现的是高性能的实时系统，就不能使用垃圾回收，这样就只剩下了 C++。

如果应用了前面这三个过滤条件之后，仍然有多种语言可供选择，我们就可以挑选生产效率最高的语言。例如，一家做 Web 应用程序的创业公司，最有可能觉得 Java、JavaScript、PHP、Python、Ruby 和 Scala 最适合解决他们的问题。如果该团队更喜欢动态类型，可能会从清单中去掉 Java 和 Scala。如果他们中有一小部分人已经熟悉 Python，并发现几个 Django 插件可以节省许多时间，Python 将成为他们的最佳选择。

5.5 选择服务器端框架

我们应该在创业公司中使用框架吗？有些程序员会告诉你框架太过笨重和复杂，所以应该使用库来代替。但是库和框架之间的区别是什么呢？通常的答案就是**控制反转**：我们把库插入到代码中并调用它们，反之我们把代码插入到框架中并让它们去调用你。这也称为**好莱坞原则**：不要给我们打电话，我们会打电话给你。如果你在开发 Web 服务，除非你调用 `socket.accept` 并编写自己的 HTTP 解析代码，否则总是应该把代码插入到某种调用你的框架中。这种框架也许是像 Ruby on Rails 这样的全栈框架，可以把控制器（controller）插入到处理请求中；或者是更加精简的框架，比如原始的 HTTP 服务器，可以插入函数去处理 HTTP 消息。无论哪种情况，你都离不开框架。

所以，选择库还是框架通常都不是问题，问题是选择**最精简的框架**还是选择**全栈框架**。全栈框架，比如 Ruby on Rails，就是一种为大多数常见任务内置提供了默认解决方案的框架，像路由、数据建模、视图渲染、国际化、配置和测试。最精简的框架，比如 Sinatra，只为你提供简单的基本功能——也许只有 HTTP 路由，再内置一点其他功能，你自己想办法处理各种常见任务。

最精简的框架可能比较合适小型的项目、原型和试验。例如，用 Sinatra 输出“Hello, World”只需要五行 Ruby 代码：

```ruby
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

如果只是个简单的小任务，容易学习、上手的框架有着巨大的优势。你只要把需要的库拿来，比如模板引擎和处理 JSON 的库，丢进去，快速地把东西做出来就好。但是，随着项目规模和重要性的提升，你将意识到自己需要有方法去处理配置、测试、安全、静态资源、监控和数据访问，就会开始把越来越多的库插入到你最精简的框架中。最终，你所做的其实就是创建一个全栈框架，只不过这个框架的所有权属于你，而且没有文档、测试或者开源社区给予支持。

> 任何库复杂到一定的程度之后，都会包含一个临时的、不合规范的、充满程序错误的、运行速度很慢的、只有一半功能的全栈 Web 框架，这是向格林斯潘的编程第十定律致敬。该定律告诉我们，任何 C 或 Fortran 程序复杂到一定程度之后，都会包含一个临时开发的、不合规范的、充满程序错误的、运行速度很慢的、只有一半功能的 Common Lisp 实现。

全栈框架包含所有内置功能只有一个原因：现实中的大部分应用程序都需要它们。即便你现在并不太用得上这些功能，框架中内置这些功能增加的成本也不多，所以因为某种程度上“感觉繁重”就舍弃不用，未免有点目光短浅。当然，并不是每一个内置的解决方案都能满足你的需求，所以要寻找 80%~90% 的默认方案都能满足要求的框架。一旦它不能满足需要，你可以立马用定制的库去代替。例如，Play Framework 是一个全栈的 Java/Scala 框架，但大多数的功能，比如数据库访问、视图渲染、缓存和国际化都是可插入式的。你甚至可以用一个单独的类去替换路由这样的核心功能：

```
public class Global extends GlobalSettings {
  @Override
  public Action onRequest(Request request, Method actionMethod) {
    return handleRequestWithCustomRoutingLogic(request);
  }
}
```

如果你正在使用 Web 框架去实现对业务至关重要的东西——不仅仅只是原型，最好的选择通常就是模块化的全栈框架。那样的话，你可以获得两个方面的好处：一是得到了一个文档完善、有社区支持的开源框架，默认功能就可以出色地处理用户的大部分需求；二是对于小部分特定的情况，也可以通过插入定制库的方法去满足要求。

为了帮助你挑选出优秀的全栈框架，我们来看看全栈框架的适用问题、数据层、视图层、测试、可扩展性、部署和安全等方面的内容。

5.5.1 适用问题

有一些 Web 框架是专用于解决特定类型的问题的。例如，Ruby on Rails 和 Django 可以简化 **CRUD 应用程序**（即在关系型数据库上执行基本的创建、读取、更新、删除操作的应用程序）的实现过程，它们内置了对数据库迁移、数据库客户端库、视图渲染、路由和脚手架（scaffolding）的支持；许多 Node.js 框架，比如 derby.js 和 express.io，都是为使用 Web sockets 的实时 Web 应用程序而设计的；DropWizard 框架则为实现 RESTful API server 进行了专门的定制，它对配置 RESTful 路由、Resource 的实现、生成 API 文档和监控有着内置的支持。如果你不能确定框架是否满足自己的需求，其实可能是没有很好地理解框架或者自己的需求，不妨进行更多研究，再多做一些原型。

5.5.2 数据层

服务器端框架大部分的任务就是解析、转换和序列化数据，所以我们要寻找能够给我们提供强大数据处理手段的框架。当你实现后台服务时，主要处理的是来自数据库的数据（这部分内容将在 5.6 节讨论）和来自客户端的数据。而客户端的数据通常是以 URL、JSON 和 XML 的形式出现的。例如，考虑下面这个 HTTP 请求：

```
Method: POST
Path: /article/5/comments
Headers: Content-Type: application/json;
Body: {userId: 10, text: "Thanks for sharing!"}
```

如果在 Ruby on Rails 中处理这个请求，就可以把下面的语句添加到 routes.rb 文件中：

```
post '/article/:articleId/comments', to: 'Comments#create'
```

然后可以创建如下控制器去处理请求：

```
class CommentsController < ApplicationController
  def create
    comment = Comment.create(
      articleId: params[:articleId],
      userId: params[:userId],
      text: params[:text])
    render :json => comment
  end
end

class Comment < ActiveRecord::Base
end
```

这里只需要几行代码，但是不妨看看 Ruby on Rails 在底层做了多少数据处理。

(1) Rails 会根据 routes.rb 中的模式去解析 URL 路径，我们才可以从 `params` 散列中提取出 `articleId`。
(2) 因为 header 的 `Content-Type` 是 `"application/json"`，Rails 会自动将请求的 body 解析为 JSON，所以我们也可以从 `params` 中抽取出 `userId` 和 `text`。
(3) 我们让 `Comment` 类扩展自 `ActiveRecord::Base`，这样就可以调用 `create` 方法将评论数据保存到数据库中。
(4) 调用 `render :json` 时，Rails 会自动将 `comment` 对象转换为 JSON，添加适当的 `Content-Type` header，并将响应发回给浏览器。

所以，我们要寻找能够让数据的处理变得轻松的框架。

5.5.3　视图层

大多数 Web 框架都提供了渲染 HTML 的模板库。当我们评估模板库的时候，主要需考虑内置的视图辅助方法、服务器端与客户端的对比、有逻辑和无逻辑模板的对比。

1. 内置视图辅助方法

正如有全栈框架一样，现在也有全栈模板库，它提供了一套实现常见视图任务的辅助方法。例如，Ruby on Rails 的 ERB 模板就提供了一些辅助方法，可以实现 i18n（国际化）、生成 URL 到应用中的控制器、生成 URL 到静态内容（例如 CSS、JS、图片）、表单渲染和模板合成（即重用布局和局部模板）。

2. 服务器端与客户端的对比

大多数的模板化技术，比如 Rails ERB 模板、Django 模板和 JSP 都是用在服务端的。服

务器从数据库中获取数据，将其填入模板中生成 HTML，再把这个 HTML 发送到网页浏览器，如图 5-2 所示。

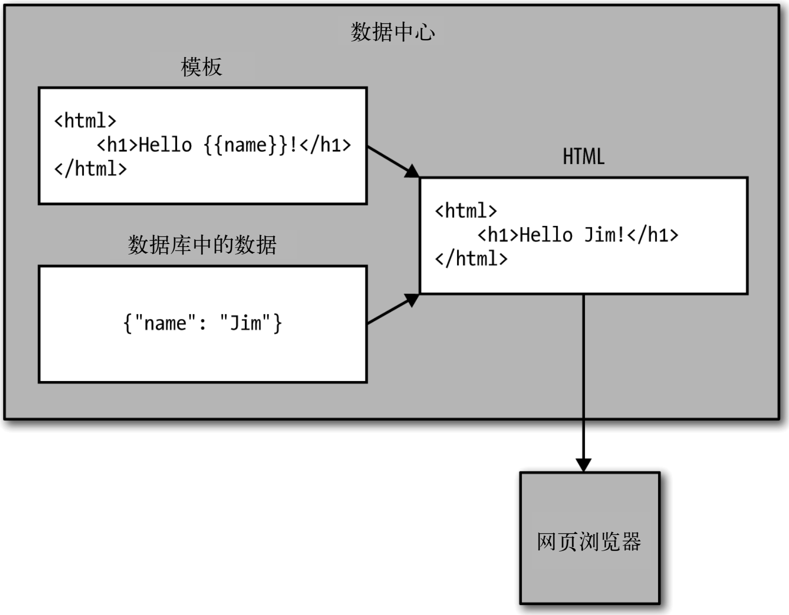


图 5-2：服务器端渲染

近年来有一种替代方案变得更加流行，就是使用可以编译成 JavaScript 的模板化技术（比如 Mustache.js），这样就可以实现在客户端进行大部分的渲染。使用这种技术仍然需要少量的服务器端渲染，才能向网页浏览器发送 HTML 页面的基本框架，但是网页中包含的从数据库中获取的数据通常被封装为 JSON，作为链接插入到 JavaScript 代码中。当浏览器执行 JavaScript 代码的时候，它会获取客户端模板，填入 JSON 数据生成 HTML，再将 HTML 注入 DOM 中，如图 5-3 所示。

客户端模板比较适合富 JavaScript 应用程序，这样的程序并不重新加载整个页面，而是通过 AJAX 去抓取数据，并使用客户端渲染重新绘制页面的一小部分。这样做也会有一些潜在的性能上的好处，因为大部分页面标记都放在 JavaScript 文件中，它们可以通过 CDN（content delivery network，内容分发网络）来提供，从而减缓网络延迟，并且可以缓存在浏览器中，所以永远不会两次加载相同的标记。不幸的是，它也同样存在着潜在的性能代价。大多数浏览器在初始页面加载的时候都为 HTML 的渲染进行了优化，但是客户端渲染需要下载 JavaScript，然后进行解析、执行，再将结果插入到 DOM 中。有些公司能够从客户端渲染中获得很好的性能，比如应用了 BigPipe 的 Facebook，但许多公司也遇到了问题，比如 Twitter，它也尝试过客户端渲染技术，但是回到服务器端渲染可以让初始页面加载时间缩短 80%。

理想的解决方案也许是同时支持在服务器和客户端渲染同样的模板。Node.js 是一个服务器端的 JavaScript 引擎，但已经成为 Web 应用的流行平台，部分原因就是它允许在服务

器端和客户端执行相同的 JavaScript 代码，该技术称为**同构 JavaScript**。例如，对于初始的页面加载，我们可以用 Node.js 在服务器端渲染 Mustache 模板，但此后对浏览器中的所有点击，都可以利用客户端渲染技术，使用相同的 Mustache 模板去重新绘制页面的部分内容。像 rendr、meteor.js 和 derby.js 这样的 Node.js 框架都努力让客户端和服务器之间的代码共享变得更加容易。

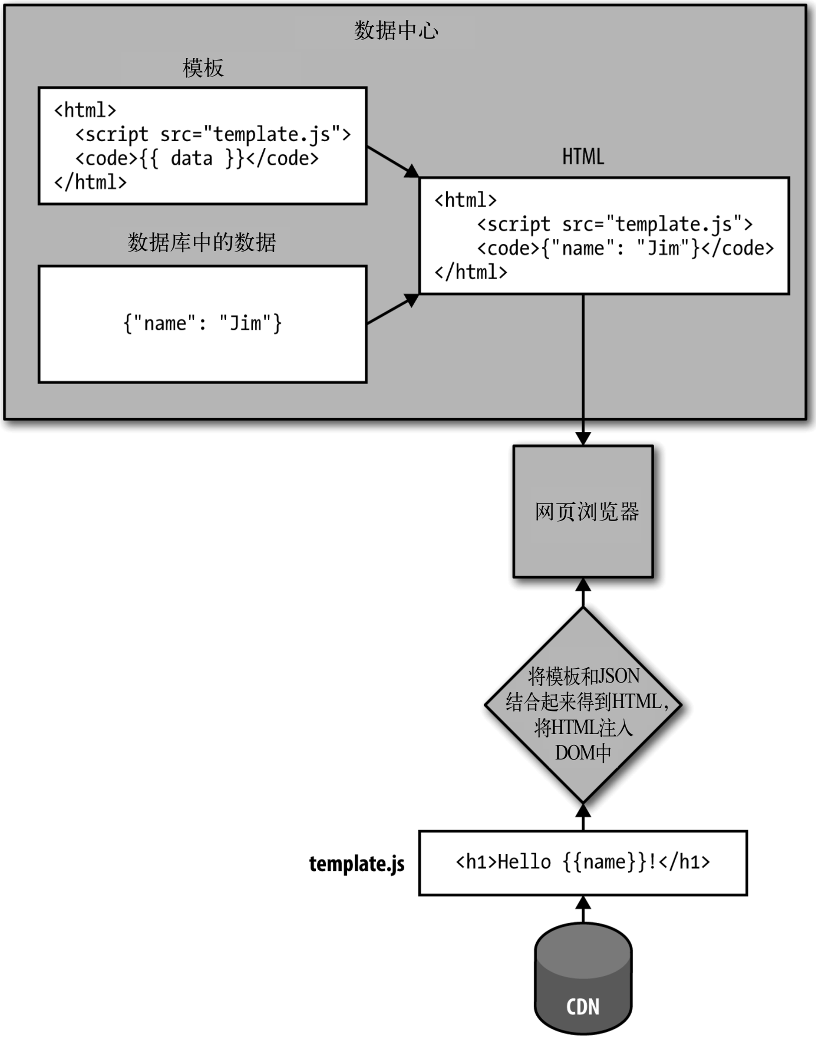


图 5-3：客户端渲染

3. 有逻辑和无逻辑模板的对比

大多数模板化语言都包含了 HTML 标记，并包含通用编程语言代码的特定语法。例如，Ruby on Rails 提供了 ERB 模板，对于放在 `<% %>` 块中的任何代码都会作为 Ruby 代码去执行：

```
<p>Regular HTML markup</p>

<%
  text = "Arbitrary Ruby code"
  puts text
%>
```

有一些程序员滥用了这一功能，把 HTML 标记、JavaScript 代码、数据库调用和其他业务逻辑都塞到一个单独的文件中（例如 my-entrice-app.php）。这样使代码维护、重用、测试和推导都变得极其困难。为了实施关注隔离，一些开发人员开始使用**无逻辑**（logic-less）模板，这样的模板专用于存放 HTML 标记，限制包含其他代码的语法。例如最流行的无逻辑模板库之一 Mustache.js，它唯一的特殊语法就是变量查找（即 `{{variable_name}}`）和基本的循环、条件（即 `{{#conditional_variable_name}}`）：

```
<p>Regular HTML markup</p>
<p>This is a {{variable}} lookup.</p>
<p>This is {{#is_enabled}}conditional text{{/is_enabled}}</p>
```

无逻辑模板的缺点就是存在**视图逻辑**（view logic）这样的东西。举例来说，在一张表格中，如果我们需要每隔一行用不同的 CSS 类名去做标记，就需要一种循环结构，可以访问索引，并检查索引是奇数还是偶数。无逻辑模板迫使我们必须实现自定义的辅助函数去处理这一类逻辑，通常会导致我们需要重新实现编程语言中辅助方法语法的大部分内容。

5.5.4　测试

在试用 Web 框架的时候，首要的事情就是为代码编写几个自动化测试（阅读 7.2.1 节了解更多信息）。如果测试的框架对于如何编写测试有着清晰的文档说明，并且提供了支持常见测试任务的库和辅助方法（例如使用内存数据库或者可以发送伪请求），这就是好的迹象。如果由于测试的框架使用了全局状态或者与低速的依赖项紧密地耦合在一起（例如编写任何测试都必须启动应用和数据库），导致难以编写单元测试，这就是不好的迹象。如果根本就没有文档说明如何编写测试，或者框架的源代码本身都没有包含许多自动化测试，这就是糟糕的迹象。

5.5.5　可扩展性

Web 框架很少成为应用程序可扩展性方面的瓶颈（阅读第 7 章了解更多信息）。所以通常来说，我们最好选择开发效率比较高，而不是处理请求比较快的框架。也就是说，如果你担心性能问题，就需要弄清楚你的应用程序是否有 I/O 或 CPU、内存方面的限制。

1. I/O限制

许多 Web 框架，比如 Ruby on Rails、Servlets 和 Django，会为每一个请求分配一个线程（或进程），在等待 I/O 调用的时候（比如等待数据库或远程 Web 服务的响应）阻塞该线程。如果线程太少，很容易出现所有线程都被占用而处于等待 I/O 的状态，阻止线程对任何新请求的处理，即便大部分线程仅仅是在空等待；如果线程太多，就会引发额外的内存使用和上下文切换导致的巨大开销。所以，确定合适的线程数量也是很困难的，因为它取决于服务器的负载和下游依赖项的延迟情况，而这一切都是经常处于变化中的。

如果你选择了错误的数值，下游依赖项的延迟只要增加一小点，就可能层叠式地传导到整个数据中心，引发延迟的增加和失败。

处理 I/O 更高效的方式是使用基于非阻塞 I/O 实现的 Web 服务端，比如 Node.js 或者 Netty。利用这种方式，在等待 I/O 完成的时候，我们并不是将线程阻塞，而是注册一个回调（或者承诺），该线程就可以继续处理其他请求。当回调被触发时，线程可以将请求唤起，完成对它的处理。因为 I/O 处理的时间会比其他进程内的处理时间长好几个量级，所以异步处理 I/O 的方式使我们对服务器资源的利用更加高效。通常来说，非阻塞服务器的每个 CPU 核心只需要一个线程（或进程），并且对下游的延迟情况也不那么敏感。

2. CPU、内存限制

尽管大多数 Web 应用都存在 I/O 限制，但如果你的服务只是执行少量的 I/O，也许就存在 CPU 限制或内存限制。这样的话，为了使性能得到最大程度的提高，你需要有能够快速执行纯计算，并且开销最小的框架（或语言）。和所有性能问题一样，找到答案的唯一方式就是进行性能测试和测量。读者可以查看 TechEmpower Web 框架基准测试作为入门，或者阅读第 7 章了解有关性能测量的内容。

5.5.6 部署

要部署代码，我们必须先解决如何构建、配置和监控的问题。

构建的过程包括代码编译、静态资源编译（例如 sass、less 和 CoffeeScript）、运行测试和封装用于生产部署的应用。有一部分构建过程也许是在开发期间进行的，比如每次刷新页面都要重新编译 CoffeeScript 文件。除非你明白如何将 Web 框架的构建过程集成到技术栈中，否则不要使用任何 Web 框架（阅读 8.3 节了解更多信息）。

构建完应用之后，就要对它进行配置以满足生产部署的需要——这时不妨寻找一些其他公司已经在用的框架，从中汲取他们的经验。例如，框架如何运行才能在单台服务器上使用所有 CPU 内核？如何才能在多台服务器上实现负载均衡？如何配置 Web 框架去处理 SSL 终端和提供静态内容，或者是否应该放到单独的 Web 服务器上去处理？大部分框架都提供了配置系统，可以对一些参数进行调整，比如 SSL 设置、静态内容设置和内存设置。我们要确保配置系统有良好的文档说明，配置能够像代码一样存放在版本控制系统中，能够提供方便的方法，可以在不同环境下使用不同的配置文件（阅读 8.4.2 节了解更多信息）。

一旦完成了代码在生产环境中的部署，就需要对其进行监控，使它持续运行。日志记录是一种最基本的监控形式，所以一定要弄清楚框架所提供的日志记录类型，确定如何调整日志级别、日志格式以及日志文件的轮换和存储。我们也要能够对一些指标进行监控，比如 QPS（每秒查询次数）、延迟、被命中的 URL、错误率、CPU 和内存占用率。有些框架可以自动显示这些指标或者通过插件来实现，有一些框架则集成了第三方服务和面板（阅读 8.5 节了解更多信息）。

5.5.7 安全

安全的实现是非常困难的。它是 5.3.4 节所列清单中的一项。我们应该使用内置了安全特性、开源且经过实际检验的框架。因为这不可能是后期再加入的功能，所以框架必须

是**默认安全的**，使我们做不安全的事情也变得困难或不可能。我们要花时间去熟悉常见的 Web 安全实践方法——开源的 Web 应用安全项目（Open Web Application Security Project，OWASP）就是一个很好的开始。本节会简短介绍一些基本知识，比如身份认证、CSRF 工具、注入攻击和安全公告。

1. 身份认证

存储密码的第一条规则就是：不要用纯文本存储密码。

*存储密码的第二条规则就是：***不要用纯文本存储密码**。

在没有深入理解如何安全存储密码之前就去存储密码是一种糟糕的想法，甚至是不道德的。这不仅对你的用户是一种风险，对于互联网上的每一个人都是如此，因为人们会在多个地方重用相同的密码。所以，不要对密码存储掉以轻心，不要自己想出一套密码存储方案，并且永远不要用纯文本存储密码。

安全存储密码至少要做到：

- 在客户端界面使用适当的密码框；
- 只通过 SSL 连接发送密码；
- 为每个密码创建长的、唯一的、随机的盐（salt）；
- 将密码和盐结合起来，使用密码安全散列函数（比如 `bcrypt`）将它们散列化；
- 保存盐和散列，将原始的密码丢弃。

在弄清楚所有这些步骤为什么是必需的，也清楚如何去实现之前，不要去碰用户的密码。除了密码之外，我们还需要了解如何安全管理会话（session）信息，包括如何正确生成会话 id、存储会话 cookies 以及处理会话过期。不自己去实现这些东西才是最佳的做法，但是我们要找到一个框架，拥有能够实现所有这些任务且经过全面测试的开源库。

2. CSRF攻击

跨站点请求伪造（Cross-Site Request Forgery，CSRF）攻击是指恶意网站让用户在受信任的网站上执行有害的动作。例如，假设你访问 win-an-ipad.com 网站，该网站让你在表单中输入一些数据并提交，这样你就有机会赢得一部 iPad。但你不知道的是这个表单实际上被提交到了 Amazon，这是一个你信任并且已经登录的网站。如果 Amazon 没有 CSRF 保护，攻击者就可以精心制作出符合要求的表单，Amazon 会将提交的表单解释为你在购买东西，因为浏览器会将你的 Amazon cookies 和提交的内容一起发送出去。

为了防范 CSRF 攻击，Web 框架应该具有为每位用户生成短暂的随机令牌的机制，将其存放在 cookie 中，如果 body 中的令牌和 cookie 中的令牌不匹配，则拒绝提交表单。这样在渲染自己网站上的合法表单时，我们就可以轻松地将令牌作为隐藏的表单字段，把它包含在表单中。当攻击者尝试渲染网站上的恶意表单时，他们无法读取你的 cookies，也就无法猜测到正确的令牌值。

3. 代码注入攻击

代码注入攻击是指恶意用户让你的应用程序执行他们的代码。如果你没有对用户生成的数据进行清洗，这是完全可能做到的。注入攻击有三种最常见的类型，分别是跨站点脚本、SQL 注入和 eval 注入。

跨站点脚本（cross-site scripting，XSS）攻击可能发生在你将未经清洗的用户生成数据放

入网页中的时候，这样攻击者就能够在页面上执行任意代码，比如偷取用户的 cookies。为了防范 XSS 攻击，我们选用的模板技术应该默认对 HTML 字符进行转义，并且在必要的时候也能够转义其他字符类（例如 JavaScript、XML）。读者可阅读 OWASP XSS 防范指南了解有关内容。

SQL 注入攻击发生在我们将未经清洗的用户生成数据放到 SQL 查询中的时候，这样攻击者可以对你的数据库进行任意的修改，比如删除所有的表。所以，要确保访问数据库的库默认会对所有的查询参数进行清洗。

eval 注入发生在将未经清洗的用户生成数据放入 `eval` 语句中时。这使得攻击者可以在服务器上运行任意代码，比如接管服务器或偷取用户数据。这里所谓的 `eval` 语句，是指任何可以接受字符串并将其作为代码执行的任意一种语言的结构。数据清洗是无法完全防范 eval 注入攻击的，所以永远不要对用户生成的数据使用 `eval` 语句[6]。事实上，`eval` 也使我们对代码的推导变得更加困难，也许还会引起一些性能上的问题，所以通常应该避免 `eval` 的使用。这一点在我们自己的代码中很容易做到，但是你永远都不知道你所依赖的框架和库是否会用到 `eval`。大多数静态类型语言都不支持任何形式的 `eval`，所以其本身就更能防范这一类的攻击[7]。

4. 安全公告

大部分流行的 Web 框架在发现了严重的安全缺陷时，都有一套通知用户的体系。目前就有 Ruby on Rails 的安全列表和 Node 安全项目。所有框架都时不时会暴露一些安全问题，所以在安全隐患被发现之后能够快速地弥补是十分关键的。我们要寻找一些能够认真对待安全问题并及时发布安全公告的框架。

5.5.8 结语

挑选框架时最主要的考虑因素就是其社区规模，因为它会影响我们招聘、寻找学习资源、利用开源库和插件的能力。下面列出了一些截至 2015 年最流行和成熟的框架，按照编程语言划分并根据首字母进行了排序，内容主要来源于 HotFrameworks 和我自己的经验。

- C#：.NET。
- Clojure：Ring、Compojure、Hoplon。
- Go：Revel、Gorilla。
- Groovy：Grails。
- Haskell：Snap、Happstack、Scotty。
- Java：Spring、Play Famework、DropWizard、JSF、Struts。
- JavaScript：express.js、sails.js、derby.js、geddy.js、koa、kraken.js、meteor。
- Perl：Mojolicious、Catalyst、Dancer。
- PHP：Laravel、Phalcon、Symfony、CakePHP、Yii、Zend。

注 6：例如，只需要使用 `characters()[]{}!+` 就可以编写任意的 JavaScript 代码。

注 7：在某些静态类型的语言中，模仿实现 `eval` 也是可能的，但必须越过重重障碍，所以极其少见。换句话说，`eval` 在动态语言中的使用要更加频繁，这也导致了严重的安全漏洞。例如 Ruby on Rails 的一个路由类内部使用了一条 `eval` 语句，使得全世界所有安装了 Ruby on Rails 的服务器都更容易受到代码注入的攻击。

- Pyhton：Django、Flask。
- Ruby：Ruby on Rails、Sinatra。
- Scala：Play Framework、Spray。

应用三个过滤条件，可以快速删减这个列表，即编程语言、适用问题和可扩展性。例如，如果你的团队更喜欢用 Java 开发，你的选项就是 Spring、Play Framework、DropWizard、JSF 和 Struts；如果你的目标是实现一个 RESTful API 服务器，Spring、Play Framework 和 DropWizard 就是最合适的；如果你知道应用会有 I/O 上的限制，Play Framework 的非阻塞 I/O 模型就是最佳选择。

如果在应用了前三个过滤条件之后，还有几个可供选择的选项，那么问题就成了挑选最适合你的部署、安全需求、数据、模板和测试需要的框架。

5.6 选择数据库

现代互联网公司要处理比以往任何时候都要多的数据。看看短短一分钟之内，互联网上生成和交换了多少数据：

- 有 100 小时的视频上传到 YouTube；
- Apple 的应用商店有 19 000 次下载；
- 有 276 000 张照片上传到 SnapChat；
- 有 350 000 条消息在 Twitter 上发布；
- Facebook 上新增了 3 000 000 个赞；
- WhatsApp 发送了 44 000 000 条消息；
- 有 204 000 000 封 email 发送了出去。

更令人惊讶的是，这些数字都在呈指数式增长——全世界生成的数据量每年都在翻番。为了处理所有这些数据，过去 15 年的数据存储系统在迅猛扩张。所以好消息就是你的创业公司有许多可供选择的数据存储选项。

而坏消息也是你的创业公司有许多存储的选择。各种术语和概念层出不穷，你是应该使用 SQL 还是 NoSQL？是模式的（schema）还是无模式的（schema-less）？ MySQL 还是 MongoDB？ Redis 还是 Riak？为了帮你回答这些问题，我先会对创业公司使用的大多数常见数据系统进行简要的介绍。接着，我会讲解挑选这些数据库所要考虑的因素，包括读取数据、写入数据、模式、可扩展性和成熟度。

5.6.1 关系型数据库

关系型数据库从 19 世纪 80 年代起就一直在数据存储解决方案中占据着统治地位。最流行的关系型数据库包括 Oracle、MySQL、PostgreSQL、MS SQL Server 和 SQLite。关系型数据库把数据存放在**表格**、**行**和**列**中。每一张表格都代表一系列有关联的项，其中每一项都放在一行，表格中的每一行都有相同的列。设想你正在做一个银行的网站，需要存放客户数据。你可以创建一张 `customers` 表，其中每一行都用 `customer_id`、`name` 和 `date_of_birth` 的一个元组代表一个客户，如表 5-2 所示。

表5-2：customers表

customer_id	name	date_of_birth
1	Brian Kim	1948-09-23
2	Karen Johnson	1989-11-18
3	Wade Feinstein	1965-02-29

关系型数据库需要定义一个**模式**去描述每一张表的结构，一般都是使用 SQL（Structured Query Language，结构化查询语言）作为数据定义语言去完成的：

```
CREATE TABLE customers (
  customer_id     INT NOT NULL PRIMARY KEY,
  name            VARCHAR(128),
  date_of_birth   DATE
);
```

模式使得关系型数据库可以实施各种**完整性约束**。例如，在上面的模式中，每一列都指定了类型（INT、VARCHAR、DATE），数据库可以利用它对每一次写入进行验证。customer_id 列还被标记为 NOT NULL PRIMARY KEY，这样数据库将会保证该列总是有值，而且每一个 customer_id 最多在表中出现一次（也就是说，该字段可以用来作为所在行的唯一标识符）。

关系型数据库也使用 SQL 作为数据操作语言。例如，下面是在 customers 表中插入一行的方法：

```
INSERT INTO customers (customer_id, name, date_of_birth)
VALUES                (1, "Brian Kim", "1948-09-23");
```

而下面这条语句则是使用 SQL 查询数据库找出名字是“Brian Kim”的客户的方法：

```
SELECT * FROM customers WHERE name = 'Brian Kim';
```

关系型数据库可以对任意一列创建**索引**，甚至还可以对多列创建**复合索引**，从而提升上述搜索查询的速度。我们也可以对多张表同时进行操作，例如假设银行中的每一位客户都有一个有余额的支票账户或储蓄账户。为了表示这些数据，可以创建一个 accounts 表，如表 5-3 所示：

表5-3：accounts表

account_id	customer_id	account_type	balance
1	1	checking	500
2	2	checking	8500
3	1	savings	2500
4	3	checking	160

注意 customer_id 是如何引用 customers 表中的 id 的。我们可以把 customer_id 列标记为**外键**：

```
CREATE TABLE accounts (
  account_id      INT NOT NULL PRIMARY KEY,
  customer_id     INT FOREIGN KEY REFERENCES customers(customer_id),
  account_type    VARCHAR(20),
  balance         INT
);
```

现在如果尝试在 accounts 表中插入一条记录，且它的 customer_id 在 customers 表中不存在，数据库将会抛出错误：

```
INSERT INTO accounts (account_id, customer_id, account_type, balance)
VALUES              (1, 555, "checking", 500)

-- 错误：无法插入或更新子行：外键约束失败
```

我们也可以使用 SQL 在多张表上进行查询，这种操作称为 JOIN。例如下面的语句能找出账户余额至少有 1000 美元的客户名称：

```
SELECT customers.name
FROM customers JOIN accounts
ON customers.customer_id = accounts.customer_id
WHERE accounts.balance > 1000
```

5.6.2 NoSQL数据库

NoSQL 表示的“Not Only SQL”（**不仅仅是 SQL**）是一个模糊的术语，用来指没有使用 SQL 的数据库，即它们并没有使用关系模型。非关系型数据库有许多类型，大部分都没有被广泛采用，比如 20 世纪 90 年代的对象数据库和 21 世纪早期的 XML 数据库。而 NoSQL 是指一种新的数据库类型，是在 21 世纪 10 年代末出现的，主要是互联网公司为了满足数据库性能、可用性和数据量上的空前需求而去努力调整关系型数据库开发的。

NoSQL 的早期灵感来自于 Google 在 2006 年发表的关于 BigTable（这是一个分布式的存储系统，设计用于处理“数以千计的商业服务器上拍字节级的数据”）的论文和 Amazon 在 2007 年发表的关于 Dynamo（这是一个高可用的键值存储系统，Amazon 的部分核心服务使用它来提供“随时在线”体验）的论文。而实际的术语“NoSQL”则是在这些论文之后才出现的，它源于 2009 年旧金山一场讨论“开源、分布式、非关系型数据库”的会议，该会议所使用的 Twitter 标签正是 #NoSQL。

该会议的描述可能是对 NoSQL 最好的定义：这是运行在数据库集群上，不使用关系模型的开源数据库。大多数常见的 NoSQL 数据库都是键 – 值存储、文档存储、面向列的数据库（column-oriented databases）和图形数据库（graph databases）。

1. 键 – 值存储

键 – 值存储是专门为单一的使用场景优化的，即根据已知的标识进行极快速地查询。它们实际上是一张分布在许多服务器间并持久化到磁盘上的散列表。流行的键 – 值存储有 Redis、DynamoDB、Riak 和 Voldemort。

大多数键 – 值存储的 API 通常都仅由两个函数组成，一个用于插入键 – 值对，一个可以根据键去查询值。下面是 Voldemort 中使用 put 和 get 函数的一个例子：

```
> put "the-key" "the-value"
> get "the-key"
version(0:1): "the-value"
```

键 – 值存储并没有使用模式，所以你可以存储任何想要的值。不幸的是，因为大部分的键 – 值存储都把值当作不透明的块来对待，所以除了根据主键查询外，它们不能支持其他任何查询机制。

2. 文档存储

文档存储与键 - 值存储类似，因为它们也是存储键 - 值对。差别就是文档存储能够察觉值的格式，所以支持更多的高级查询功能。流行的文档存储有 MongoDB、CouchDB 和 Couchbase。

我们简单地看一个使用 MongoDB 的例子。MongoDB 可以把 JSON **文档**存储在**集合**中，多少类似于关系型数据库把**行**存储在**表**中。MongoDB 对于文档没有预先定义的模式，所以可以存储任何想要的 JSON 数据。例如，下面的语句使用 `save` 命令将一个 JSON 文档存放在一个叫 `people` 的集合中：

```
> db.people.save(
{_id: "the-key", name: "Ann", age: 14, locationId: 123})
```

在 MongoDB 中，每一个文档都有一个叫 `_id` 的字段，用来作为键。在上面的例子中，`_id` 被明确地设置为"`the-key`"，但你也可以让 MongoDB 为你自动生成 `_id` 字段，只要在调用 `save` 的时候不指定就可以了：

```
> db.people.save({name: "Bob", age: 35, locationId: 456})
```

现在，你可以使用 `find` 命令看到 `people` 集合中的所有文档：

```
> db.people.find()
{"_id": "the-key", "age": 14, "name": "Ann", "locationId": 123}
{"_id": ObjectId("545bdc1e"), "age": 35, "name": "Bob", "locationId": 456}
```

你也可以通过 `id` 去查找特定的文档，效率和键 - 值存储几乎是一样的：

```
> db.people.find({"_id": "the-key"})
{"_id" : "the-key", "age": 14, "name": "Ann", "locationId": 123}
```

和键 - 值存储不同的是，文档数据库也可以通过文档内的任何一个字段去执行查询。例如，下面展示了如何查找所有 `name` 字段被设置为"`Ann`"的文档：

```
> db.people.find({"name": "Ann"})
{"_id": "the-key", "age": 14, "name": "Ann", "locationId": 123}
```

许多文档数据库甚至还支持对文档内的字段进行索引，就是所谓的**辅助索引**，让搜索变得更快。不幸的是，文档数据库通常不支持 JOIN 查询。例如，假设有另一个叫 `locations` 的集合：

```
> db.locations.find()
{"_id": 123, "city": "Boston", "state": "Massachusetts"}
{"_id": 456, "city": "Palo Alto", "state": "California"}
```

要取出某一个人和他们所居住城市的名称，唯一的办法就是在应用程序代码中使用两个连续的查询：一个从 `people` 集合中取出数据，再用第二个从 `locations` 集合中取出位置数据。还有一种替代的方法就是对数据进行反规范化，我们不是把 `locationId` 存储在 `people` 集合的每个文档中，而是直接存储 `city` 和 `state`。这么做使得读取数据更简化、速度更快，因为只需要一条查询即可，但是会让更新变得更加复杂、更容易出错，速度更慢，因为存在冗余数据。例如，如果城市被重新命名，就必须用新的名称去更新 `people` 集合中的每一个条目，而不是仅仅更新 `locations` 集合中的一个条目即可。

3. 面向列的数据库

主流的面向列的数据库有 HBase 和 Cassandra。它们表面上和关系型数据库类似，因为也是把数据存放在表中，也是由行和列构成。Cassandra 甚至还有表的架构，拥有和 SQL 非常类似的查询语言，叫 CQL。但它们主要的差别就在于关系型数据库通常都是**面向行的**，意味着它们为许多数据行的操作进行了优化；而面向列的数据库则是对许多列的操作进行了优化。例如，考虑表 5-4 所示的 `books` 表。

表5-4：books表

id	title	genre	year_published
1	Clean Code	tech	2008
2	Code Complete	tech	1993
3	The Giver	sci-fi	1993

这些数据是如何存储在硬盘上的呢？在关系型数据库中，每一行的值都会被放在一起，所以从概念上讲，序列化之后的数据看起来也许像这样：

```
1:Clean Code,tech,2008;2:Code Complete,tech,1993;3:The Giver,sci-fi,1993;
```

从中可以看到，一行中所有的列都是连续排列的。可以对比一下面向列的存储是如何序列化相同数据的：

```
Clean Code:1,Code Complete:2,The Giver:3;tech:1,2,sci-fi:3;2008:1,1993:2,3;
```

以这种形式存储的话，一列中所有的值都连续排列，以列值作为键（即计算机书名），以 `id` 作为值（即 1、2）。现在看看下面这条查询语句：

```
SELECT * FROM books WHERE year_published = 1993;
```

由于这条查询使用了 `SELECT *`，所以需要读取任何匹配行的每一列。硬盘在连续读取时性能最佳，所以这条查询在面向行的存储中效率最佳，因为一行中所有的列都是紧挨着的。接下来再比较下面这条查询：

```
SELECT COUNT(*) FROM books WHERE year_published = 1993;
```

由于这条查询使用了 `SELECT COUNT(*)`，因此它只需要读取 `year_published` 列的值。这样的聚合查询在面向列的数据库中更有效率，因为一列中所有的值都是紧挨着的。

4. 图形数据库

图形数据库把数据表示为用有向边连接的节点。虽然其他的 NoSQL 数据存储主要是为了满足在集群上运行的需求，但大多数的图形数据库是运行在单节点上的，是为了满足高效存储、查询和导航关系型数据的需求的。主流的图形数据库有 Neo4j 和 Titan。

考虑如图 5-4 所示的示意图形。

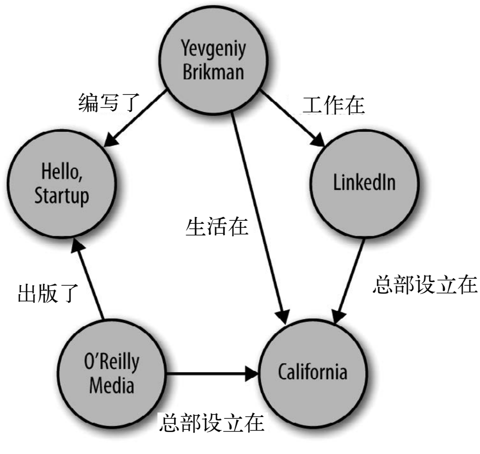


图 5-4：示意图形

要在 Neo4j 中创建该图形的节点，可以使用 CREATE 命令：

```
CREATE
  (yevgeniy { name: "Yevgeniy Brikman" }),
  (oreilly { name: "O'Reilly Media", industry: "publishing" }),
  (california { name: "California" }),
  (linkedin { name: "LinkedIn", industry: "social networking" }),
  (hello { name: "Hello Startup", yearPublished: 2015 })
```

还可以使用 CREATE 将节点用边连接起来：

```
CREATE
  (yevgeniy)-[:WROTE]->(hello),
  (yevgeniy)-[:LIVES_IN]->(california),
  (yevgeniy)-[:WORKED_AT]->(linkedin),
  (oreilly)-[:PUBLISHED]->(hello),
  (oreilly)-[:HEADQUARTERED_IN]->(california),
  (linkedin)-[:HEADQUARTERED_IN]->(california)
```

这里没有预先定义的模式，所以节点和边可以用任意属性创建。你可以使用 MATCH 命令根据这些属性去查询图形，比如寻找 name 属性设置为“Yevgeniy Brikman”的节点的查询命令如下：

```
MATCH (person)
WHERE person.name = "Yevgeniy Brikman"
RETURN person
```

该查询会返回一个节点：

```
(8 {name:"Yevgeniy Brikman"})
```

我们也可以查询节点之间的关系。例如，下面的语句是找出所有总部设立在加利福尼亚的公司：

```
MATCH (company)-[:HEADQUARTERED_IN]->(location { name: "California" })
RETURN company
```

该语句执行后将得到两个节点：

```
(9 {name:"O'Reilly Media", industry: "publishing"})
(10 {name:"LinkedIn", industry: "social networking"})
```

甚至还可以更进一步，比如查找由居住在加利福尼亚的人所写的书：

```
MATCH (book)<-[:WROTE]-(person)-[:LIVES_IN]->(location { name: "California" })
RETURN book
```

将得到一个节点：

```
(11 {name:"Hello, Startup", yearPublished: 2015})
```

5.6.3 读取数据

在上面对关系型数据库和 NoSQL 数据库的介绍中，我们看到了不同查询模型下各种数据库的优缺点，如表 5-5 所示。关系型和图形数据库对于通用目的的数据存储来说是较好的选择，因为它们较为灵活的查询模型可以处理大多数创业公司不断变化的访问模式。其他的 NoSQL 数据库对于具有特定目的的数据存储来说是不错的选择，因为它们满足了一些特定的访问模式。

表5-5：查询模型

数据库类型	访问模式	JOIN支持	索引
关系型	非常灵活的查询模型	是	主键、次键、组合
键 - 值	主键查询	否	主键
文档	主键、次键查询	否	主键、次键
面向列	单列操作	否	主键、次键
图形	非常灵活的查询模型	是	主键、次键

在读取数据时，我们还要考虑另一个因素，即数据是如何呈现的。例如，看看下面这个 Java 类：

```
public class Person {
  private long id;
  private String name;
  private int age;
  private List<String> skills;
}
```

我们可以很容易地把 `Person` 类表示为一个类似的 JSON 文档：

```
{
  "_id": 123,
  "name": "Linda",
  "age": 35,
  "skills": ["Java", "Scala", "Ruby"]
}
```

对于键 - 值存储或文档数据库，我们可以仅用一条命令去保存或检索此 JSON 文档：

```
> db.people.save(
{_id: 123, name: "Linda", age: 35, skills: ["Java", "Scala", "Ruby"]})
> db.people.find({_id: 123})
{_id: 123, name: "Linda", age: 35, skills: ["Java", "Scala", "Ruby"]}
```

另一方面，这些数据在关系型数据库中的规范化表示稍微有点不同，如表 5-6、表 5-7 和表 5-8 所示。

表5-6：people表

person_id	name	age
12345	Linda	35

表5-7：skills表

skill_id	skill_name
1	Java
2	Scala
3	Ruby

表5-8：people_skills表

person_id	skill_id
12345	1
12345	2
12345	3

如果要从 Person 对象中获取数据，我们需要用到连接（JOIN）三张表的查询：

```
SELECT people.person_id, people.name, people.age, skills.skill_name
FROM people
  JOIN people_skills ON people.person_id = people_skills.person_id
  JOIN skills ON skills.skill_id = people_skills.skill_id
WHERE people.person_id = 12345
```

该查询将返回三行，如表 5-9 所示，这时我们必须仔细地把它们解析到 Person 类的各个字段中。

表5-9：查询结果

person_id	name	age	skill_name
12345	Linda	35	Java
12345	Linda	35	Scala
12345	Linda	35	Ruby

即便是一个简单的类，从关系型数据库映射到内存中的表示也是很复杂的，这就是所谓的**阻抗失配**。许多**对象关系映射**（Object Relational Mapping，ORM）工具都是为了尝试解决这一问题而诞生的，比如 ActiveRecord 和 Hibernate，但通常都会引起争论——有人责备这些工具会暴露抽象泄露（leaky abstractions）问题，还会引发一些性能问题。这并非个别 ORM 工具存在的问题，而是由于映射本身就是一个难题。所有你能想出的解决方案都会包含一些严重、痛苦的取舍。

对于许多创业公司而言，在应用程序规模较小、性能需求较低的时候，使用 ORM 是值得的，它可以正常满足 80%~90% 的用户需求。重要的是要认识到，使用 ORM 并不意味着可以忽略关系型数据库底层的工作细节。我们仍然需要弄清楚关系型数据的建模、规范化、索引、链接和查询调优等知识，这样才能知道如何正确地存储数据，并满足 ORM 难以实现的那 10%~20% 的用户需求。

随着创业公司的成长，ORM 成功的概率可能会下降到 80% 以下。这时有两个选择：要么放弃对象（去掉“O”），要么放弃关系型数据库（去掉“R”），这样也就不会再面临映射的问题。如果选择第一个选项，就可以用关系型或者函数式模型在内存中表示数据，从而取代对象。例如，可以使用**函数式关系映射程序**（Function Relational Mapper，FRM），比如 Typesafe Slick，代替 ORM；如果选择了第二个选项，就可以从关系型数据库切换到 NoSQL 存储。正如上面所介绍的，用键 – 值或文档存储表示的数据和内存中的表示相似，所以映射问题也比较容易解决。唯一不好的选择就是尝试编写自己的 ORM，得到的肯定是比已经发展多年的开源 ORM 工具更差的解决方案。

5.6.4　写入数据

大多数 NoSQL 数据库都为处理**聚合**而进行了优化，即键 – 值存储中的一个值、文档存储中的一份文档、面向列的数据库中的一列[8]。虽然写入一个聚合通常并不复杂，也可以保证原子性，但尝试写入多个聚合时，NoSQL 数据库无法保证原子性。

假设你正在开发一个银行网站，需要存储每一个账户的余额，如表 5-10 所示。

表5-10：accounts表

account_id	balance
1	500
2	8500
3	2500

像 Voldemort 这样的键 – 值存储中，插入一个值（一个聚合）是很容易的：

```
> put "4" "5,000"
```

但更新已有的数据却比较困难。要更新银行的账户余额，我们可能需要先发出请求获取当前的值，在应用程序代码中计算新的值，然后发出第二个请求，保存新值（要注意这一操作并不是原子性的）[9]。

在文档数据库中，更新一个账户（一个聚合）会更简单。例如，要从账户中减去 100 美元，我们可以使用 MongoDB 的 `update` 函数，以及增量操作符 `$inc`：

```
> db.accounts.update({_id: 1}, {$inc: {balance: -100}})
```

在关系型数据库中，更新一个账户也很简单：

注 8：术语“聚合”出自 *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*，图形数据库并没有专为聚合而优化，所以不在讨论中。

注 9：有一些键 – 值存储，比如 Redis 和 Riak，对值的数据类型的支持是有限的，比如 `string`、`integer`、`set`、`list` 和 `map`。这使得我们能够在一次事务处理中进行某些类型的更新，比如用 Redis 中的 `INCR` 命令让一个整数增 1。

```
UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1
```

但是，如果这家银行要收取 100 美元的年费，要如何从每个账户中减去 100 美元呢？在 MongoDB 中，我们可以将 `multi` 选项设置为 `true` 来实现：

```
> db.accounts.update({}, {$inc: {balance: -100}}, {multi: true})
```

问题在于，对多个账户（多个聚合）的更新并不是原子性的。拥有多个账户的客户在打开银行网站时，可能会看到其中的一个账户已经扣除了 100 美元，但另一个账户却没有扣除。在关系型数据库中，这种情况是不可能发生的，因为所有更新都是原子性的：

```
UPDATE accounts
SET balance = balance - 100
```

当你需要事务性语义时，这甚至会成为一个更严重的问题。例如，如果要从账户 1 转 100 美元到账户 2，在大多数 NoSQL 数据库中，需要发出两条单独的命令：

```
> db.accounts.update({_id: 1}, {$inc: {balance: -100}})
> db.accounts.update({_id: 2}, {$inc: {balance: 100}})
```

问题是这两条更新的发生并不是原子性的，如果这期间有什么地方出错会怎样呢？例如，如果从账户 1 将钱扣除，将钱添加到账户 2 之前，数据库却崩溃了，钱就凭空消失了。如果要在 NoSQL 数据库中进行原子性的更新，就必须在应用程序代码中手动实现两阶段的提交，这是复杂且容易出错的。

如果把这些更新封装在一个事务中，大多数的关系型数据库都会自动解决这一问题：

```
START TRANSACTION;
  UPDATE accounts
  SET balance = balance - 100
  WHERE account_id = 1;

  UPDATE accounts
  SET balance = balance + 100
  WHERE account_id = 2;
COMMIT;
```

这样就可以在多行，甚至在多张表上执行多个更新，这些更新要么全部成功，要么全部都会回滚。

5.6.5　模式

大多数 NoSQL 数据库在自我宣传时都声称是**无模式的**，但是关系型数据库却需要我们提前定义好模式。这种差别会对我们造成一点误导。虽然 NoSQL 数据库也许并不关注数据的模式，但在某些时候，应用程序必须知道数据的格式才能对它进行读取。例如，考虑下面从 MongoDB 中读取文档的 Java 代码：

```
DBCollection books = db.getCollection("books");
BasicDBObject query = new BasicDBObject();
query.put("author", "Yevgeniy Brikman");
```

```
DBObject book = books.findOne(query);

String title = (String) book.get("title");
int pages = ((Number) book.get("pages")).intValue();
Date datePublished = (Date) book.get("datePublished");
```

为了发送有意义的查询命令并对结果进行解析，Java 代码必须知道字段的名称（author、title、pages 和 datePublished）以及这些字段的类型（String、int 和 Date）。这就是模式！换句话说，这并不是模式和无模式的问题，而是模式是否是显式并由数据库应用，或者是否是隐式而由应用程序代码应用的问题。

让数据库应用模式有助于自动防止一大类错误的出现，其作用类似于编程语言中的静态类型。例如，关系型数据库可以确保你的查询不会出现表名或列名的拼写错误，也不会把错误的数据类型存入到列中，字符串也不会超过预定义的大小限制，主键 id 会是唯一的，外键指向的也会是其他表中有效的 id。在 NoSQL 领域也同样需要这些完整性的检查，但我们必须在应用程序代码中手动去实现。让经过充分测试的关系型数据库帮你自动实现这一切通常会更加安全，特别是这样可以保证模式集中在一个地方应用，而不是分散在直接与 NoSQL 数据库交互的所有应用程序代码中。这种模式对于想要弄清楚所处理的数据类型的开发者来说，也起到了文档的作用。

无模式对两种情况的处理是有利的，第一种情况就是需要存储无结构或不统一的数据的时候。例如，用户生成数据、事件跟踪数据和日志信息也许是不规则或不可预料的格式。如果把这样的数据存放在关系型数据库中，最终会出现大量的 NULL 列、一些名称没什么意义的列（例如 col1、col2、col3）或者一些存储“二进制大数据块”的列（例如把一个 JSON 文档放到一列中），这些情况在关系型领域中都属于反模式。

第二种情况就是在进行数据迁移的时候。为了改变存储在关系型数据库中的数据类型，我们不但要更新应用程序代码，还要更新模式。根据数据库以及数据量的不同，在处理数据列或表的添加、删除或者完整性约束上都可能会有昂贵的代价，想要不下线而完成任务是难以实现的。对于 NoSQL 数据库，我们所要做的就是更新一下应用程序代码，让它能够处理新旧两种数据格式，这样迁移就算完成了。或者，更确切地说，迁移仅仅是开始，它会随着新数据的写入而增量式地发生。例如，如果之前把图书的数据存放在 MongoDB 中，然后将 pages 字段重命名为 pageCount 字段，就必须对 Java 代码进行更新，如下所示：

```
int pages;

if (book.containsKey("pages")) {
  pages = ((Number) book.get("pages")).intValue();
} else {
  pages = ((Number) book.get("pageCount")).intValue();
}
```

这种方式使增量式的、零下线时间的迁移变得更容易，因为我们可以同时处理数据库中所有现存图书的旧字段名和要写入数据库的新书的新字段名。但经过几次迁移之后，这一类的 if 语句会让应用程序代码变得难以维护。因此，我们可能得做一些额外的工作，生成能够加速这一迁移的后台脚本，以便在代码变得乱七八糟之前对旧格式的代码进行清理。

5.6.6 可扩展性

可扩展性并不是选择 NoSQL 的主要动机之一 [10]。像 Google 和 Amazon 这样的公司面对的可用性和性能上的需求已经超出了任何一台单独的服务器的能力范围，他们在**垂直扩展**（即为单台服务器增加更多的 RAM 或 CPU）上已经达到了极限，所以他们需要能够运行在服务器集群上的系统，实现**水平扩展**（即添加更多的服务器）。

一旦数据存储从单台服务器走向多台服务器，就会涉及**分布式系统**。所有的分布式系统都受到 CAP 定理的约束，该定理表述如下。

> 一个分布式系统不可能同时满足以下三点：
>
> - 一致性（所有节点能够同时访问同一份数据）；
> - 可用性（保证每一个请求都会接收到成功或失败的响应）；
> - 分区容忍性（系统一部分出现任意信息丢失或故障时，系统仍能继续工作）。

在一致性（**C**onsistency）、可用性（**A**vailability）和分区容忍性（**P**artition Tolerance）中，只能择其二。在实践中，总是会有服务器故障或网络丢失信息的情况，所以所有的分布式系统必须选择 P——也就是说，不可能牺牲分区容忍性。所以实际的问题就成了，当存在网络分区的时候，是要坚持一致性还是坚持可用性？

有些系统，比如 MongoDB、HBase 和 Redis，总是尝试在所有节点上保持数据的一致性，所以在存在网络分区的情况下，也许会失去可用性。其他一些系统，比如 Voldemort、Cassandra、Riak 和 CouchDB，可以实现**最终一致性**，意味着当存在网络分区的时候，他们会保持可用性，但不同的节点也许会存在不同的数据，这样的一些冲突将会在后续得到解决。事实上，即便没有分区，分布式系统中的数据传播也肯定是要花时间的，所以即便在正常的操作期间，最终一致的系统也会在不同的节点存在着不同的数据，至少在很短暂的时间内有这样的情况存在。

实现可水平扩展的分布式数据系统有两个主要策略，即复制和分区。

1. 复制

复制（replication）是指将相同的数据复制（copy）到多台服务器或多个**副本**上。复制的一个主要好处就是容错性，服务器和硬盘随时都会发生故障，所以不论你选择什么样的数据库技术，都需要保证数据的副本不只存放在一个地方，以防停机和数据丢失。我们至少要把数据复制到一个**备用副本**中，这样的副本并不能提供任何实时的服务，它只能在主数据库下线的时候被调换出来使用。你也可以将数据复制到一个或多个**活动副本**中，这样的副本可以提供实时的服务，所以我们可以增加更多的这种副本，水平地扩展数据库。通过复制实现可扩展性有两种常用的方法，即**主 - 从复制**和**多主复制**。

在主 - 从复制中，如图 5-5 所示，所有写入操作就发生在单一的节点上（主节点），该节点会将这些改变传播到一个或多个副本上（从节点），所有的读取操作都是在副本上进行的。

注 10：除了图形数据库。我们选用图形数据库的目的与其他的不同，是为了高效存储和查询关系型数据的需要。

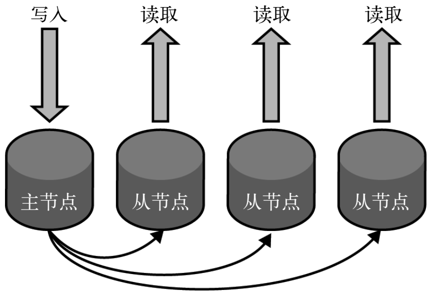


图 5-5：主 – 从复制

在多主复制中，如图 5-6 所示，所有的节点都是平等的，所以它们可以接受读取和写入，并把变化传播到所有的同级节点中。

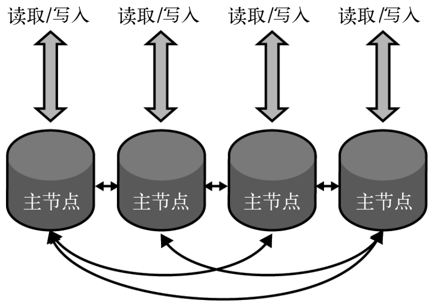


图 5-6：多主复制

主 – 从复制主要为了满足更多读取请求的扩展需要，而多主复制能够同时满足读取和写入请求的扩展需要。那么，为什么不在所有场景中都使用多主复制呢？答案是复杂性。只有单个写入点（单个主节点）的系统更容易进行推导和维护，而且由于大多数数据库遇到的读取请求会大大超过写入请求，主 – 从复制足以应付许多扩展性方面的挑战。然而，如果单个主节点成为了写入操作的瓶颈，多主复制所带来的复杂性也许就是值得的。

这样的复杂性源于有多个写入点的系统可能同时会看到两个不同的节点对同一条数据出现了不同的更新。这就是所谓的**冲突**，使用多主复制的系统必须实现**冲突解决策略**。在某些情况下，把写入操作合并在一起是可行的。例如，Amazon 使用 Dynamo 数据库来存储购物车数据，这种情况下把冲突的写入操作合并在一起就是可行的，因为只需把用户已经放到购物车中的所有东西表示出来就可以了。其他的一些策略，比如**后写胜出**，具有较新时间戳或向量时钟的值将会覆盖较旧的值；还有**用户指定**，即所有的冲突版本都会被保存下来，而客户端代码在读取这些数据的时候必须判断哪些是要保留的。

2. 分区

复制是将相同的数据复制到多台服务器上，而分区则是将数据的不同子集复制到不同的服务器上。分区的目的是让数据集在 n 台服务器之间分配，这样每一台服务器都只需要承担负载的 $1/n$。由此，我们可以添加更多的节点实现水平扩展，进一步降低每台服务器的负载。然而，如果没有将数据正确地分区，某个节点将会承受比其他节点更多的负载，这就形成了所谓的**热点**，将成为扩展工作的一个瓶颈。为了避免这样的瓶颈，我们需要

选择合适的分区策略。有两种主要的策略可供选择：**垂直分区**和**水平分区**。

垂直分区是指将无关联的数据类型拆分开，比如将一些列移动到单独的表中或者将表移动到单独的数据库中。例如，假设我们正在开发一个银行网站，该网站有两个页面：一个用来管理储蓄账户，一个用来管理支票账户。最初设计的时候，我们可能会把所有的用户数据存放在一张名为 `users` 的表中，如表 5-11 所示。

表5-11：users表

user_id	username
1	alice123
2	bob456
3	jondoe

还会把所有的账户数据存放在一张名为 `accounts` 的表中，如表 5-12 所示。

表5-12：accounts表

account_id	user_id	type	balance
1	1	checking	100
2	1	savings	500
3	2	checking	1500
4	3	savings	250

如果网站大受欢迎，`accouts` 表会变得非常大，导致数据库难以承受负载的增加。一种解决方案就是根据支票账户和储蓄账户，垂直地将数据划分到两个单独的表 `checking`（如表 5-13 所示）和表 `savings`（如表 5-14 所示）中。

表5-13：checking表

account_id	user_id	balance
1	1	100
3	2	1500

表5-14：savings表

account_id	user_id	balance
2	1	500
4	3	250

你可以把这些表存放在单独的数据库中，如图 5-7 所示。

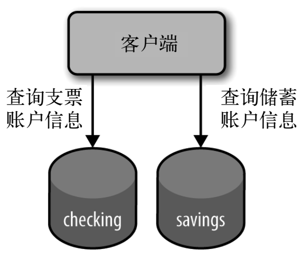


图 5-7：垂直分区

由于网站的每一个页面要么展示支票账户，要么展示储蓄账户，不会同时展示两种账户，因此现在每一个数据库只需要处理一部分请求即可。此外，每一个数据库也不需要和其他数据库去竞争 CPU、内存和磁盘空间。然而，分区总是会有代价的。例如，我们无法再进行 JOIN 操作。如果要获取给定的 `user_id` 的用户名和总余额，我们就无法使用带 JOIN 操作的单条查询，而是不得不发出三次独立请求，给所有分区而不是一个分区带来负载。我们也无法再自动应用外键约束，就像 `checking` 和 `savings` 表中的 `user_id` 列。如果一张单独的表变得太大，垂直分区也无济于事。例如，如果这个银行网站变得非常受欢迎，仅仅一张 `checking` 表可能就会把单台服务器拖垮。

还有一种方法就是使用水平分区（又称为**分片**，sharding），利用这种方式可以把一张数据表中的行划分到不同的分区（又称为**片**，shards）中。例如，如果银行网站有 10 台服务器和 100 万名用户，我们就可以根据 `user_id` 对原始的 `accounts` 和 `users` 表进行分区，让 `user_id` 在 0~100000 的数据存放在服务器 0 上，100001~200000 的数据存放在服务器 1 上，以此类推，一直存放到服务器 9，如图 5-8 所示。

该策略对于需要展示单个用户数据的页面有很好的表现，因为所有的数据都存放在一个单独的片中。例如，对于垂直分区模式，我们需要三次请求才能获取一个 `user_id` 的用户名和账户总余额，而对于水平分区模式，我们只需要发送一条查询给该用户的分片即可：

```
SELECT users.username, SUM(accounts.balance)
FROM users JOIN accounts ON users.user_id = accounts.user_id
WHERE users.user_id = 100455
```

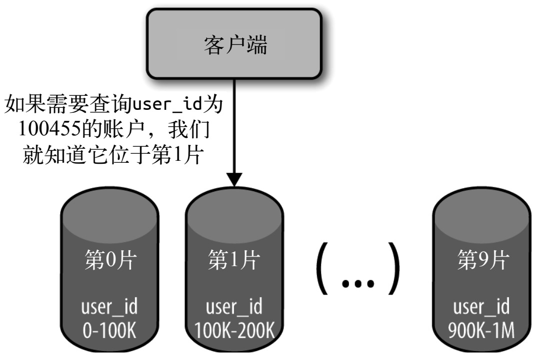


图 5-8：水平分区

如果大多数请求只是一次针对一个用户，那么每个分片只会承担 10% 的负载，我们也仍然能够使用 JOIN 和外键约束。如果用户数据继续增长，我们还可以添加更多的分片来实现水平扩展。

但是，该方法也存在一些严重的缺陷。例如，如果想要找到账户中至少有 500 美元的所有用户，情况又会怎样呢？我们没有办法提前知道这些用户都在哪个分片中，所以只能执行**分散聚集查询**，对所有分区进行查询，这样会给所有分片带来负载，而不仅仅是一个分片。水平分区也使得 id 的生成变得更加复杂。对于单个数据库而言，我们可以很容易地使用递增计数器；但对于多个数据库，就必须进行额外的处理才能避免发生冲突。更麻烦的是，我们的数据和访问模式也将随着时间的推移发生变化，所以，最开始的分

区策略可能会在日后成为必须关注的问题。对分区策略的更改，即所谓的**再平衡**，将是困难且昂贵的，因为它需要对大量的数据进行迁移。

5.6.7 故障模式

每一种数据存储，不论是手动分片的 MySQL 部署，还是自动分片的 MondoDB 集群，在某一时刻都会发生故障。问题是这些系统存在多少种故障模式，弄清楚和解决每一种故障的难度有多大？通常，比较简单的方案修复起来也比较容易，许多 NoSQL 方案是非常复杂的，特别是那些支持多点写入、自动分片、自动再平衡的方案。

Pinterest 在网站的每月页面访问次数从零到上百亿的发展过程中，不得不放弃 MongoDB 和 Cassandra 的一个主要原因就是大量不同故障模式的复杂性问题。例如，如果 NoSQL 存储中的自动平衡算法出现了 bug 会发生什么？一种可能的结果就是复制操作会受到阻碍，最后导致一致变成了永远不一致。另一个可能的结果是集群变得不再平衡，即便你有十个结点，所有的流量也会跑到其中一个上。第三种可能的结果就是出问题的再平衡算法会将坏数据散布到所有的节点中，所有数据完全崩溃。而最糟糕的情况是，当你打算更新 NoSQL 软件的新版本去修复这些问题时，你又揭开了另一个失败模式：在分布式系统中，所有节点相互之间都必须进行通信（比如具备多主复制的 NoSQL 存储），如果新版软件所使用的通信协议有不兼容的变化，想要升级集群而不让一切停下来将是不可能的。

5.6.8 成熟度

公司的数据是你业务中最重要的一部分。它存在的时间极可能比任何功能、任何应用，甚至公司本身都要长。你可以改变编程语言、Web 框架，甚至重写代码一百次，但是你所收集的数据是不变的。你需要让这些数据四处迁移（例如将数据填入数据仓库、Hadoop 集群或者搜索索引中），对数据进行备份、监控，这一切没有成熟的工具生态系统是难以实现的。因此，我们需要以一种安全、可靠并且支持良好的方式去存储数据，数据才能够长期保存下来。著名的投资者乔治·索罗斯说过一句很有名的话，“好的投资是无聊的”。我想说好的数据存储也是无聊的。

> 我们早期在 GitHub 尝试过许多不同的数据库，但是在过去二三年，我们已经把它们都拿掉，只剩下了 MySQL。MySQL 已经发展了大概 20 年，它也有让人讨厌的地方，但我们知道是哪些。它相当稳定，最重要的是，我们知道如何对它进行扩展。自公司创办以来，我们一直都在进行扩展，所以这是一个已知的因素。
>
> 我们一直尝试去掉越来越多的东西，尽可能简化我们的栈。新的数据库也许很吸引人，但在稳定性上就未必了。正常运行就是人们喜欢的，能正常运行就已经了不起。所以我们一直都对我们所选择的技术感到越来越无聊，却也高兴得不能再高兴了。
>
> ——Zach Holman，GitHub 软件工程师

数据存储技术花了很长时间才走向成熟。看看早期发布的一些最流行的关系型数据库：Oracle 出现在 20 世纪 70 年代、Microsoft SQL Server 出现在 1989 年、MySQL 和 PostgreSQL 出现在 1995 年。这些数据库已经持续发展了 20~40 年，仍然致力于安全性、

可靠性和性能上的提升。数据存储不是一个可以快速解决的问题，所以建议大家遵循以下经验法则。

> 通用的数据存储技术需要十年才能走向成熟[11]。

在我 2014 年写这本书的时候，NoSQL 存储的平均年龄是 6 岁。以下列出了一些主流 NoSQL 存储的发布日期：CouchDB 诞生于 2005 年、HBase 是 2006 年、Neo4j 是 2007 年、Cassandra 是 2008 年，而 MongoDB、Redis、Riak 和 Voldemort 则于 2009 年问世。许多公司都报告了因 NoSQL 不成熟而带来的问题。例如，MongoDB 多年来一直是争论的源头，人们声称它的容错机制在设计上是有问题的，许多公司为此已不再使用它，包括 Pinterest、Urban Airship、Etsy、Viber 和 Bump。但是这种情况并不单单在 MongoDB 身上出现，Twitter、Facebook 和 Pinterest 也放弃了 Cassandra；Instagram 和 Viber 则放弃了 Redis；Signal Engage 和 Canonical 则不得不放弃 CouchDB。这并不意味着 NoSQL 数据库就是糟糕的选择，而是说它们是更有风险的选择，而数据存储通常不应该是我们要冒风险的地方。

5.6.9 结语

当我们挑选数据存储技术的时候，最重要的就是考虑它的成熟度。你可以解决编程语言或客户端框架上存在的限制，但无法解决数据丢失的问题。但从成熟度这一点看，关系型数据库应该是我们做出任何数据存储决定的默认选择。我们可以先用关系型数据库对问题进行建模，看看在碰壁之前我们可以走多远。如果很长时间都没有碰壁也不用惊讶，因为关系型数据库也是相当灵活的。例如，我们可以使用标准的关系型模式和完整性约束，利用查询语言支持索引、事务和 JOIN 操作的优点。我们也可以把它们当作无模式的键 – 值存储，或者当作方形、星形或雪花形的离线分析存储，甚至作为一种快速的 JSON 文档存储[12]。

如果使用关系型数据库遇到了障碍，极有可能是因为我们的数据量和可用性方面的需求已经超出了单台服务器的能力范围。此时，我们要优先考虑找出可以扩展的最简单的解决方案。按照复杂程度，以下列出了最常见的一些选择：

- 对数据存储格式和现有数据库的查询进行优化；
- 在数据库之前设置缓存（例如内存缓存）；
- 建立主 – 从复制；
- 对无关联的表进行垂直分区；
- 对单张表进行水平分区；
- 建立多主复制。

一般而言，我们要尽可能避免对数据进行分区，并坚持使用单点写入。分片的多主系统存在更多的故障模式，在实现 JOIN 操作、事务、强制完整性约束、迁移、更新、备份和 id 生成方面也会更加复杂。NoSQL 的问题则是，你不得不在上述各个方面做出牺牲，即

注 11：也许不仅仅是数据存储，任何复杂的软件要走向成熟都需要花这么长的时间。正如 Joel Spolsky 写的，“十年得一好软件，请习惯这一点。”

注 12：PostgreSQL 和 MySQL 对 JSON 文档提供了原生支持，PostgreSQL 版也许比 MongoDB 的速度还要快。

便大多数用户场景对于这一切并不是必要的。而即便在确有必要做出牺牲的情况下，用关系型数据库去实现也可以让我们走得很远——Facebook 的集群有 4000 个 MySQL 节点，每秒可以处理 6000 万次查询。也就是说，虽然关系型数据库原本是为在单台服务器上运行而设计的、大多数 NoSQL 数据库则是为专门在集群上运行而设计的，但它们通常都提供了内置的工具，可以在一致性和可用性、复制因子和分区数量之间进行权衡调整。某些情况下，这也意味着 NoSQL 存储对需求而言将是最简单的解决方案。

未来技术的发展有两种趋势值得关注。第一个趋势是 NoSQL 生态系统正变得更加成熟。随着时间的推移，许多 bug 将会修复，可用性也将提升，而且我们也将很好地理解每一种数据存储技术在不同行业的优势和劣势，所以 NoSQL 也许会在越来越多的用户场景下成为最简单的解决方案。第二个趋势是 **NewSQL 数据库**的出现，这种数据存储技术为在集群上运行而设计，却仍然支持关系型模型（模式、SQL、JOIN、索引和事务）。这种类型的数据库大概在 2011 年开始出现，所以在成熟度上甚至还比不上 NoSQL 数据库，但我们不妨看看它们将如何发展，该技术应用的例子包括 Google Spanner、VoltDB、FoundationDB 和 Clustrix。

5.7 小结

如果你正在创业，对技术栈的初始选择很简单：选择你所了解的。尽可能地去使用开源和商业技术，只有在代表公司“秘密武器”的部分才要内部去实现。如果你足够幸运，公司的发展已经度过了最初的阶段，你的选择就会变得有点复杂。

下面是在评估编程语言时所要考虑的关键取舍。

编程范式

　　它是面向对象的，还是函数式编程语言？它是否支持静态类型或者自动内存管理？

适用问题

　　例如，C 对于嵌入式系统是特别合适的，Erlang 适合容错的分布式系统，而 R 则适合统计。

性能

　　该语言对并发的处理能力如何？该语言是否使用了垃圾收集？如何对收集器进行调整？

生产效率

　　该语言的流行程度如何？针对该语言的框架和库的数量有多少？它的简洁程度如何？

下面是评估服务器端框架时需要考虑的关键取舍。

适用问题

　　例如，Rails 特别适合 CRUD 类型的应用程序、DropWizard 适合 RESTful API 服务器、Node.js 适用实时的 Web 应用。

数据层

　　该框架是否能帮助你处理 URL、JSON 和 XML ？

视图层

该框架是否有许多内置的模板辅助方法？它使用的是服务器端渲染，还是客户端渲染？它使用的是有逻辑还是无逻辑的模板？

测试

为构建在该框架之上的应用编写单元测试是否很容易？该框架本身是否经过充分测试？

可扩展性

该框架使用的是阻塞还是非阻塞 I/O ？你是否根据使用场景对框架进行了性能测试？

开发

你是否知道如何将框架集成到你的开发中？是否知道如何在生产过程中配置、部署或监控框架？

安全

该框架是否提供了充分测试的方法去处理认证、CSRF、代码注入和安全公告？

下面是评估数据库时要考虑的关键取舍。

数据库类型

它是关系型数据库，还是 NoSQL 存储（键－值存储、文档存储、面向列的数据库或者图形数据库）？

读取数据

你是否需要通过主键或次键去查询数据？是否需要 JOIN 操作？如何将数据映射到内存中的表示？

写入数据

写入更新数据时仅仅是一个聚合，还是多个聚合？是否需要原子性的更新或事务？

模式

该模式是显式存储在数据库中，还是隐式存储在应用程序代码中？你的数据是统一的还是无结构的？

可扩展性

仅仅通过垂直扩展数据库能否满足需求？如果不行，数据库是否支持复制、分区，还是两者都支持？

故障模式

系统可能出现多少种故障方式？调试故障的困难程度如何？

成熟度

该数据库已经发展了多长时间了？有多少公司正在使用它？该数据库的支持工具生态系统有多丰富？

最后，我们在考虑技术栈时要把它作为一个整体。值得一提的是，许多公司已经不再使用单独、统一的技术栈（这样的技术栈对所有使用场景都应用单独的一种编程语言、框架和数据存储方案），而是朝着一种**多语言编程**模型发展（可以针对不同的使用场景应用不同的技术）。也就是说，我们基于若干不同的服务去实现一个技术栈，而每一种服务都

是使用不同的语言和框架编写的，并通过远程消息与其他服务通信（阅读 7.2.2 节了解更多信息）。例如，电子商务网站可能是由以下几部分构成的：

- 一个用 JavaScript 和 Node.js 实现的前端框架，通过非阻塞的 JSON-over-HTTP 调用从后台获取数据；
- 一个用 Python 和 Flask 实现的 RESTful 后台服务，将产品和用户数据存放在 PostgreSQL 中；
- 一个用 Java、DropWizard、Lucene 和 Redis 实现的 RESTful 后台服务，负责管理搜索索引；
- 一个 HBase 集群用于离线分析。

这种多语言模型的优点在于可以针对每一个任务使用最佳的工具，对代码进行隔离，实现组件的松耦合。然而，需要学习、部署和维护这么多不同的技术，这也是特别巨大的开销。所以，这种方法通常只对那些一个统一的代码库已不能满足扩展需要的大公司才有帮助。读者可以阅读 7.2.2 节更深入地了解其中的各种利弊。

第 6 章

整洁的代码

6.1 代码是给人阅读的

> 编程是一种让他人了解你想让电脑做什么的艺术。
>
> ——Donald Knuth

作为一名程序员，只有不到 50% 的工作时间是花在编程任务上的。编程的时间里面，阅读代码和编写代码的时间比大大超过了 10∶1，而实际花在编写代码的极少部分的时间中，80% 以上的时间又是在维护代码，即修改或修复已有的代码。如果一天工作 8 小时，能有 5 分钟花在编写新代码上就已经不错了。结果就是，程序员的工作并不是在编写代码，而是在理解代码。

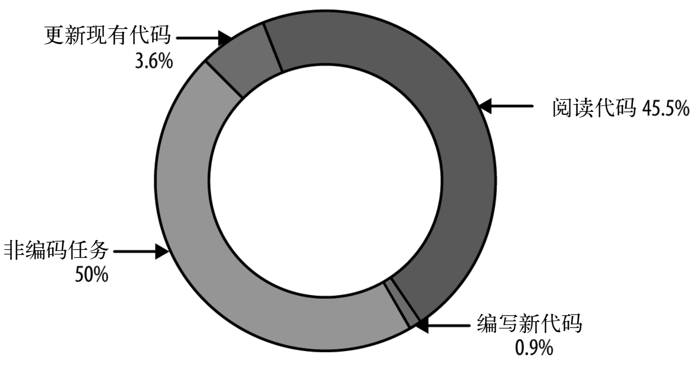


图 6-1：开发人员时间分布

这就是为什么整洁的代码至关重要。所谓**整洁的代码**，是指代码专为人的理解而优化。记住，创业和人是密不可分的，所以对代码来说，最重要的并不是运行得多快或者使用什么样的算法，而是它对使用它的人有什么样的影响。编写整洁的代码并不是为了理想

主义，也不是因为有些书上说你必须这么做（即便不是本书），更不是因为空格比制表符更优美，而是因为作为程序员，你要把大部分时间花在理解和维护代码上，这只是为了让自己方便。

我们来做个试验。假设你看到了下面这段 Java 代码：

```
public class BP {
public void cvt(File i,File o) {
BufferedReader r=null;
BufferedWriter w;
String l,j="[";
String[] p;
try {
r=new BufferedReader(new FileReader(i));
} catch (FileNotFoundException e) {}
try{
while ((l=r.readLine())!=null){
p=l.split(",");
if(!p[3].equals("fiction")&&!p[3].equals("nonfiction"))continue;
j+="{";
j+="title:\""+p[0]+"\",";
j+="author:\""+p[1]+"\",";
j+="pages:\""+Integer.parseInt(p[2])+"\",";
j+="category:\""+p[3]+"\"";
j+="},";
}
try {
r.close();
} catch(IOException e) {}
} catch(IOException e) {}
j+="]";
try {
w=new BufferedWriter(new FileWriter(o));
w.write(j);
w.close();
} catch (IOException e) {}}}
```

这段代码是干什么的？不知道的话真得停下来几秒钟，看看能不能读懂。这段代码不到 30 行，功能其实非常简单。你读懂了吗？

我敢打赌你刚刚只是瞥了一眼代码就立即放弃了。因为这只是一本书中的代码，而不是你的实际工作，你有权利这样做，但有时候你就不能回避了。你会在工作中碰到这样的代码，知道那是已经不在公司的某个人写的，也没有什么文档，那段代码还负责处理业务中一个关键部分。它到处都是 bug，而你的任务就是去修复它。到了这个时候，你就会认识到整洁代码的重要性了。

本章将介绍实现整洁代码的一些基本原则，包括代码布局、命名、不要重复自己（don't repeat yourself，DRY）、单一职责原则（single responsibility principle，SRP）、函数式编程、松耦合、高内聚、注释和重构。我不会论述有关整洁代码的每一个主题，也不会涵盖所有的细微差别，更不会过多关注理论上的东西。相反，这只是一份实践指南，指导你解决创业中可能遇到的最常见的代码质量问题。

本章开头的示例代码揭示了我们将要介绍的几乎所有问题。我们在阅读的过程中，将会

找到每一个问题的解决方案，用它们去改进这段示例代码，最终得到一段易于理解的代码。你在本章看到的解决方案并不适用于所有用户场景，但是同样的问题却会到处出现，所以本章的目标就是通过一些具体的例子，帮助大家识别出这些问题。换句话说，请做好阅读一大堆代码的准备。

6.2 代码布局

在阅读过程中，你会期望本书的编写应当遵循某些规则：字通过句号组成句子，句子通过分行组成段落，段落通过标题组成章节，小结标题使用较大的字体，引文是缩进的，而与主题关系不大的讨论则作为脚注出现在页面底部。

> 语法工具存在这么多个世纪并不是偶然，它们满足了读者的需求和潜意识的要求。
>
> ——William Zinsser，《写作法宝》

尽管编程不像写作有那么长的历史，但代码的读者也期望你能遵循一些代码布局的规则，能清晰表现程序是如何组织的。例如，对于大多数大括号语言（例如 Java、C、JavaScript），读者期望你将代码块放在大括号里，每一块内容都缩进，并用新行把函数隔开。如果你违背了这些规则，读者的生产效率和理解能力将会急剧下降。

> 格式编排的基础理论表明，良好的视觉布局可以展示程序的逻辑结构。让代码看起来漂亮肯定是值得的，但更宝贵的是可以展示代码的结构。
>
> ——Steve Macconnell，《代码大全》

看看如果代码布局错误会发生什么。2014 年 2 月，Apple 的 Safari 网页浏览器被发现在验证 SSL 服务器秘钥的过程中存在一个巨大的安全漏洞：

```
static OSStatus SSLVerifySignedServerKeyExchange(
  SSLContext *ctx,
  bool isRsa,
  SSLBuffer signedParams,
  uint8_t *signature,
  UInt16 signatureLen)
{
  OSStatus        err;
  ...

  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
  ...

  err = sslRawVerify(...);

fail:
  SSLFreeBuffer(&signedHashes);
```

```
    SSLFreeBuffer(&hashCtx);
    return err;
  }
```

你发现 bug 在哪里了吗？就是一行中的两个 goto 语句：

```
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
```

这段代码有两个布局错误。第一，它没有把每一条 if 语句的主体放在大括号中。第二，缩进是错误的，会让人误以为两条 goto 语句都是 if 语句的主体，而实际上只有第一条才是。如果你修复了这两个布局错误，这个 bug 就更明显了：

```
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
    goto fail;
  }

  goto fail;
```

第二条 goto 语句总是会被执行，它跳过语句下方所有的检查（包括关键的 sslRawVerify 检查），跳转去执行 fail 标签下的内容：

```
  fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

fail 标签返回了 err，后者此时被设置为零，告诉调用程序所有的检查均已通过，虽然实际上并没有检查。这个 bug 被取了一个很形象的别名——“gotofail”，它导致数以百万的 iOS 和 OS X 设备受到中间人攻击的威胁。布局并不是这段代码唯一的问题，而代码布局的 bug 通常不会那么严重，但它却很好地提醒了我们，代码布局的重要性不在于让代码好看，更主要的是让代码的结构变得一目了然。

你的团队应该强制实施一整套代码布局的约定，包括空白、新行、缩进和大括号的使用。尽管程序员喜欢讨论空格好还是制表符好、大括号应该放在哪里这些问题，但实际的选择并不是太重要，真正要紧的是在代码库中要保持一致。大多数文本编辑器、IDE 都提供了设置格式的工具，许多版本控制系统也提供代码提交前的检查工具，帮助我们应用一些常见的代码布局。

布局约定的一个很好的例子是 Google 的 Java 风格指南。你可以利用它对本章开头代码段的布局进行改进：

```
  public class BP {
    public void cvt(File i, File o) {
      BufferedReader r = null;
      BufferedWriter w;
      String l, j = "[";
      String[] p;

      try {
        r = new BufferedReader(new FileReader(i));
      } catch (FileNotFoundException e) {}
```

```
      try {
        while ((l = r.readLine()) != null) {

          p = l.split(",");
          if (!p[3].equals("fiction") && !p[3].equals("nonfiction")) {
            continue;
          }

          j += "{";
          j += "title:\"" + p[0] + "\",";
          j += "author:\"" + p[1] + "\",";
          j += "pages:\"" + Integer.parseInt(p[2]) + "\",";
          j += "category:\"" + p[3] + "\"";
          j += "},";
        }

        try {
          r.close();
        } catch (IOException e) {}
      } catch (IOException e) {}

      j += "]";

      try {
        w = new BufferedWriter(new FileWriter(o));
        w.write(j);
        w.close();
      } catch (IOException e) {}
    }
  }
```

只需要在正确的地方加上一些空白，代码的结构就初步显现了：这是一个叫 BP 的类，有一个叫 cvt 的方法，似乎在读入一个叫 i 的文件，并在生成一个叫 j 的 String 时对该文件的内容进行遍历，然后把 j 写入到一个叫 o 的文件。你可以再多理解一点，但如果要更进一步，就需要有更好的变量名、函数名和类名。

6.3　命名

每一个代码库都定义了自己的语言，它是由类名、方法名、变量名、函数名、包名、文件名和目录名组成的。如果代码布局是语法的话，代码中的名称就是单词，它们是你用来思考代码的言语。好的名称应该能回答所有的重要问题，应该精确、全面，能够揭示意图，并且遵循约定。

6.3.1　回答所有重要的问题

> 变量、函数或类的名称应该回答所有重要的问题。它应该告诉你它存在的原因、它是干什么的，以及如何使用。
>
> ——Robert C. Martin，《整洁代码之道》

本书开头的代码段就是不可理解的，因为它没有名称或概念可供你的大脑领会：

```
public class BP {
  public void cvt(File i, File o) {
    BufferedReader r = null;
    BufferedWriter w;
    String l, j = "[";
    String[] p;
```

太短的名称（即 i、o、r、j）传递不出任何信息。缩写的名称（即 BP、cvt）在你写代码的时候也许是可以理解的，但是为什么要强迫别人在阅读你的代码时费劲去破译这些名称呢？因为开发人员阅读代码的时间本来就比编写代码的时间多，如果还要花时间去弄清楚一个神秘的名称，这种差距就更明显了。而少输入几个字符所节省的时间却是微不足道的，在现代文本编辑器和 IDE 都支持自动完成的情况下就更是如此。我们应该让名称需要多长就多长，可以回答代码是什么、为什么和怎么样的问题。

下面是改进名称的第一次尝试：

```
public class BookParser {

  public void convert(File inputCsv, File out) {
    BufferedReader reader = null;
    BufferedWriter writer;
    String tmp, data = "[";
    String[] parts;
```

这个小的改变应该能够让你更好地理解代码。从类名 BookParser 中，可以猜到它是用来解析有关图书的数据的；从方法名 convert 和参数名 inputCsv 和 out 中，我们知道它应该是将 CSV 格式的图书数据转换成某种类型的输出格式。

6.3.2　要精确

我们要挑选能够精确描述“该段代码是做什么”和“为什么做”的名称。例如，我们应该精确地使用对应的词，如果有一个叫 open() 的方法，那么对应的方法就应该叫 close()；如果有一个 input，通常也应该有一个 output。convert 函数在对应方面就不是很一致，它有一个读参数叫 inputCsv，而写参数则叫 out。我们可以把它们命名为 inputCsv 和 outputJson，让参数的目的更加清晰：

```
public void convert(File inputCsv, File outputJson) {
```

现在，你可以猜到该函数的作用是将图书数据从 CSV 格式转化为 JSON 格式了。我们可以利用这一知识，把变量 reader 和 writer 重命名为 csvReader 和 jsonWriter，让它们的名称更加精确：

```
BufferedReader csvReader = null;
BufferedWriter jsonWriter;
```

我们也应该避免使用一些不明确的变量名称，比如 tmp、data 和 parts：

```
String tmp, data = "[";
String[] parts;
```

这些名称太过一般，我们可以用它们去存储任何类型的数据，所以对于“变量是用来干什么的”和“为什么会存在”并没有任何提示，不过就是比之前单个字母的名称稍好一

点。如果把这几个变量相应地重命名为 line、json 和 fields，代码会变得更加清晰：

```
String line, json = "[";
String[] fields;
```

我们一定要明智地选择用词，肯定有比 temp、num 和 data 这样不明确的术语更好的选择。而且相对数字符号，我们更应该选择单词，即不要用 subtotal1 和 subtotal2 这样的名称，而是选用能够清楚表示值的含义的词，比如 subtotalWithShipping 和 subtotalWithShippingAndTax。一定要想出一个好的词，哪怕查词典也可以。

6.3.3 要全面

名称应该能全面反映是什么、为什么和怎么样这三个问题。BookParser 类中的名称比之前要好一些，但仍然丢失了一些细节。例如，方法名 convert 并没有告诉你它转换的是什么，更加全面的名称应该是 convertCsvToJson：

```
public void convertCsvToJson(File inputCsv, File outputJson) {
```

如果力求全面导致名称长得可笑，通常也是一个信号，表明你所命名的对象承担了太多的职责（阅读 6.6 节了解更多信息）。例如，如果 BookParser 的代码不仅将 CSV 文件转换为 JSON，还要删除 CSV 文件并生成一份报告，你就必须把名称改成 convertCsvToJsonAndDeleteCsvfileAndGenerateReport 这样的形式。这个名称这么长，就已经表明代码所做的事情太多了。解决的办法并不是使用缩写或首字母组合，而是把这些职责分解到三个独立的函数中，每个都有自己的名称，比如 convertCsvToJson、cleanupCsvFile 和 generateReport。

我们在命名时也应该全面体现变量、函数或类应该怎么用。例如，line 和 fields 变量中应该存储什么数据并不明显，这一行行的是什么？是一行行代码，一行行音符，还是一根根钓鱼线？如果你把变量相应地命名为 csvLine 和 csvFields，一切就明了了：

```
String csvLine, json = "[";
String[] csvFields;
```

更加全面的名称不仅让代码的阅读变得更加容易，还有助于防止 bug 的出现。举个例子，我们看看下面这句代码：

```
double totalWeight = packagingWeight +
                     (itemWeight * numberOfItems);
```

发现什么地方有问题吗？如果相同的一行代码，但是变量名不同，又能发现什么：

```
double totalWeightInLbs = packagingWeightInLbs +
                          (itemWeightInKgs * numberOfItems);
```

很明显，这里有一个 bug，因为你不应该把千克（itemWeightInKg）和磅（packagingWeightInLbs）混在一个计算公式中。通过全面命名，可以将信息编码在变量名中，让错误的代码看起来就是错误的[1]。在变量名中包含单位只是一个例子（例如 lengthInMeters

注 1：这一说法出自 Joel Spolsky 的一篇文章，名为“Making Wrong Code Look Wrong”，这也是匈牙利命名法最初的意图所在。

就比只用 length 更好），所有能表示变量应该如何使用的命名对我们都是有帮助的（例如 csvLine 比只用 line 更好）。

要注意的是，变量名中所编入的信息不应该和类型系统所编入的任何东西产生冗余（如果你使用的是静态类型的编程语言）。例如，以下名称就是有冗余的：

```
String csvLineString; // 冗余的变量名
```

变量名中的单词“String”是不需要的，因为变量的类型已经被指定为 String，所以编译器会自动为你强加上去。另一方面，totalWeightInLbs 中的 InLbs 和 itemWeightInKgs 中的 InKgs 并不是多余的，因为这两个变量都是 doubles 型的，类型系统无法表示出它们所代表的不同的测量单位。

6.3.4　揭示意图

好的命名应该能够揭示意图。计算机只会关心代码是干什么的，而人却会关心代码**为什么**要这么做。例如，BookParser 的代码到处都是各种数字，至于为什么出现这些数字却没有任何提示：

```
if (!csvFields[3].equals("fiction") &&
    !csvFields[3].equals("nonfiction")) {
  continue;
}

json += "{";
json += "title:\"" + csvFields[0] + "\",";
json += "author:\"" + csvFields[1] + "\",";
json += "pages:\"" + Integer.parseInt(csvFields[2]) + "\",";
json += "category:\"" + csvFields[3] + "\"";
json += "},";
```

csvFields 的第三个下标有什么特殊的？为什么要将 csvFields[3] 分别和“fiction”及“nonfiction”做比较？ csvFields[0]、csvFields[1] 等都存放了什么？把这些**魔法数字**替换成命名的常量是个不错的主意。在 Java 中定义常量的方法之一就是使用一个 enum：

```
public enum CsvColumns {
  TITLE, AUTHOR, PAGES, CATEGORY
}
```

在一个 enum 中，每一个常量都有一个名称以及根据其定义的顺序得到的序号。例如，CsvColumns.TITLE.ordinal() 为 0 而 CsvColumns.PAGES.ordinal() 为 2。下面的代码将 BookParser 中的魔法数字用 CsvColumns enum 替换：

```
String title = csvFields[TITLE.ordinal()];
String author = csvFields[AUTHOR.ordinal()];
Integer pages = Integer.parseInt(csvFields[PAGES.ordinal()]);
String category = csvFields[CATEGORY.ordinal()];

if (!category.equals("fiction") && !category.equals("nonfiction")) {
  continue;
}

json += "{";
```

```
json += "title:\"" + title + "\",";
json += "author:\"" + author + "\",";
json += "pages:\"" + pages + "\",";
json += "category:\"" + category + "\"";
json += "},";
```

采用更好的名称后，我们就可以更清晰地看出 CSV 文件的每一行都包含了 4 列（title、author、pages 和 category），而 category 列的值必须是“fiction”或者“nonfiction”。这也给我们启发，可以针对 category 列采用另一个 `enum`：

```
public enum Category {
  fiction, nonfiction
}
```

我们可以使用 `valueOf` 函数自动将一个 `String` 转化为相应的 `enum` 值（如果匹配的值没有找到的话则抛出异常），所以不再需要专门将 `category` 和“fiction”“nonfiction”这两个值做比较：

```
Category category = Category.valueOf(csvFields[CATEGORY.ordinal()]);
```

6.3.5 遵循约定

比遵循特定的命名规则更加重要的是，要在整个项目中始终如一地遵循这些规则。比如，你会不会把变量在一个地方命名为 `recordNum`，在另一个地方又命名为 `numRecords`？你会不会一会儿把接口叫作 `PaymentProcessor`，一会儿又叫 `IPaymentProcessor`？有时把实现命名为 `CreditCardPaymentProcessor`，有时又命名为 `CreditCardPaymentProcessorImpl`？你是不是正在把设计模式的一些共享词汇，比如 factory、builder、decorator 和 visitor 这些词用到面向对象编程上，把 monad、iteratee、reader 和 lens 这些词用到函数式编程上？

在大多数情况下，我们都应该遵循编程语言的约定，例如在阅读 Ruby 代码时，就应该知道它的约定是对方法和变量名采用蛇形命名法（例如 `my_method_name` 和 `my_variable_name`）、对类名采用骆驼命名法（例如 `MyClassName`）、对常量采用大写的蛇形命名法（例如 `MY_CONSTANT_NAME`）；如果方法返回的是布尔类型则在其名称后面加上问号（例如 `is_empty?`）。如果该方法会引发突然情况、有副作用或者有破坏性，就在方法名后面加一个感叹号（例如 `fire_missles!`）。我们应该为自己的团队建立一系列的编码约定，包括命名的规则，把它们写下来，在整个代码库中强制执行。

> 如果我在看一个由 10 名工程师写的文件，应该让我几乎无法区分哪部分是哪个人写的才对。对我来说，这就是整洁的代码，而达到这一目标的方法就是通过代码评审[2]以及发布你的风格指南、模式和语言惯用语。只要你做到这一点，每个人都会变得更有生产效率，因为所有人都知道如何以相同的方式去编写代码。到了那个阶段，你们主要关注的就是在写什么，而不是怎么写的问题了。
>
> ——Nick Dellamaggiore，LinkedIn 和 Coursera 软件工程师

注 2：阅读 7.2.3 节了解更多信息。

6.3.6 命名真难

> 计算机科学只有两件难事：缓存失效和命名[3]。
>
> ——Phil Karlton

好的名称在很大程度上与所处的环境有关。通常来说，在想出好的名称之前，你必须要先成为你产品领域、技术栈、团队文化习俗方面的专家。我在编写新的函数时，有时候不得不先用 foo 这样的没什么意义的名称，先实现函数体，再做几次测试，只有在我足够理解它的用途之后才会起一个合理的名称。有时候，即便那样的名称也是不合适的。几天之后，只要我对问题领域有更深入的理解，我就得回过头来改一个更好的名称。命名是如此重要，所以我们在这一章会好几次回过头去修改前面的名称。但现在，我们先谈论一下错误处理。

6.4 错误处理

BookParser 的代码会默默吞下各种错误，比如在尝试读取 inputCsv 的时候会有一个空的 catch 块：

```
try {
  csvReader = new BufferedReader(new FileReader(inputCsv));
} catch (FileNotFoundException e) {}
```

如果把 JSON 写入磁盘的时候出现了错误，它也会静静地把异常吞下：

```
try {
  jsonWriter = new BufferedWriter(new FileWriter(outputJson));
  jsonWriter.write(json);
  jsonWriter.close();
} catch (IOException e) {}
```

如果我们运行这段代码而没有生成 JSON 文件，缺乏恰当的错误处理会使调试变得很困难。我们弄不清楚代码是在读取 CSV 文件时失败、在 CSV 中没有有效的记录、JSON 文件的路径是无效的，还是硬盘空间已经不足。每一个程序都有不同的错误处理需求，但是不应该是静静地吞下各种错误。

清晰的错误消息是整洁代码的主要特征。我们可以抛出异常、把错误消息作为返回值的一部分，或者将错误记录在日志中——只要不是静静地发生失败就行了。由于 BookParser 是对业务数据进行处理，我们通常要让数据尽可能保持原样，所以如果遇到了任何类型的错误，就应该让整个转换过程明显地抛出失败信息。为此，我们可以把所有无用的 try/catch 块去掉，让异常可以向上传导给调用者：

```
public void convertCsvToJson(File inputCsv,
                             File outputJson) throws IOException {
  try (
    BufferedReader csvReader =
      new BufferedReader(new FileReader(inputCsv));
    BufferedWriter jsonWriter =
      new BufferedWriter(new FileWriter(outputJson))
```

注 3：还可以这么说："计算机科学只有两件难事：缓存失效、命名和差一错误。"

```
) {
  String csvLine, json = "[";
  String[] csvFields;

  while ((csvLine = csvReader.readLine()) != null) {
    csvFields = csvLine.split(",");

    String title = csvFields[TITLE.ordinal()];
    String author = csvFields[AUTHOR.ordinal()];
    Integer pages =
      Integer.parseInt(csvFields[PAGES.ordinal()]);
    Category category =
      Category.valueOf(csvFields[CATEGORY.ordinal()]);

    json += "{";
    json += "title:\"" + title + "\",";
    json += "author:\"" + author + "\",";
    json += "pages:\"" + pages + "\",";
    json += "category:\"" + category + "\"";
    json += "},";
  }

  json += "]";

  jsonWriter.write(json);
}
}
```

唯一留下的 try 块就是 try-with-resource 语句，它可以保证打开的用于读取和写入（csvReader 和 jsonWriter）的文件将被正确地关闭，哪怕抛出了异常。这会使 convertCsvToJson 可以更容易地调试，因为所有错误都将被明显地报告出来；而且也更容易阅读，因为不再需要到处放 try/catch 块了。

6.5 不要重复自己

> 系统中的每一项知识都必须具有单一、无歧义、权威的表示。
>
> ——Andrew Hunt、David Thomas，《程序员修炼之道》

避免重复是实现整洁代码最根本的原则之一。有点讽刺的是，避免重复的思想不断以不同的名称重复出现，比如不要重复自己（DRY）、单点真理、一次并且只有一次。重复可能出现在技术实现的任何一个地方，包括架构、代码、测试、过程、需求和文档，它可能是以下几个原因引起的。

- 我们需要用多种方式表示相同的信息，比如在数据库模式、数据库访问层、HTML 标记和 CSS 中列出相同的列。
- 语言限制：比如在 Java 中指定 getter 和 setter。
- 缺少反规范化：比如从数据库的表中导出数据。
- 缺少时间，导致需要复制和粘贴代码。
- 没有意识到这个问题，比如多个开发人员在一个大型代码库中创建自己的 StringUtil 类，因为他们不知道类似的类已经存在（也因为他们不知道开源库甚至有更好的版本）。

重复不仅因为要多次实现相同的事情而浪费了时间，而且还妨碍了我们对代码的理解和维护。如果代码不 DRY，那么我们每需要回答一个有关代码问题，就可能必须去查看多个地方；我们每需要做修改，就必须确保不会错过任何一个副本；如果有一个副本不同步，就会导致矛盾和 bug 的出现。

如果发现自己在一次次地编写相同的代码，或者只是做一小点改变就会涉及代码库中的一半内容，那么就要想想办法让代码变得更加 DRY。特别当我们必须一次次地重复相同的**过程**，那么就需要实现一种自动化的过程；如果不只在一个地方有相同的**逻辑**，那么就需要实现抽象，以便共享单一的实现。

举个例子，来看看 `BookParser` 代码是如何构造 JSON 的：

```
String csvLine, json = "[";
while ((csvLine = csvReader.readLine()) != null) {

  // ...

  json += "{";
  json += "title:\"" + title + "\",";
  json += "author:\"" + author + "\",";
  json += "pages:\"" + pages + "\",";
  json += "category:\"" + category + "\"";
  json += "},";

}

json += "]";
```

这里有很多重复：将 JSON 元素放在括号（[] 或 {}）中的代码出现了多次，在 JSON 对象中创建键 – 值项的代码也出现了多次。所有这些重复导致了几个 bug，你发现了吗？其中一个是经典的复制 / 粘贴错误，`pages` 变量是一个 `Integer`，在插入 JSON 的时候不应该放在引号中：

```
json += "pages:\"" + pages + "\",";
```

第二个 bug 是 JSON 数组最后一个元素的后面多了一个逗号：

```
json += "},";
```

我们可以修复这些 bug，但最大的重复仍然没有去掉：JSON 和 CSV 是通用的数据格式，我们没理由从头编写代码去处理它们。**重新发明轮子**是最常见和不必要的重复，只要有可能，就应该使用开源库去代替（阅读 5.3 节了解更多信息）。例如，可以使用 Java 的 Jackson 库，让代码更加 DRY 和稳定。因为 Java 是基于类的面向对象语言，表示数据的规范做法就是创建一个类：

```
public class Book {
  private String title;
  private String author;
  private int pages;
  private Category category;

  //（省略了构造函数和getter方法）
}
```

可以构造出 Book 对象的一个 List，代替手动生成 JSON 的 String：

```
List<Book> books = new ArrayList<>();

while ((csvLine = csvReader.readLine()) != null) {
  csvFields = csvLine.split(",");

  String title = csvFields[TITLE.ordinal()];
  String author = csvFields[AUTHOR.ordinal()];
  Integer pages =
    Integer.parseInt(csvFields[PAGES.ordinal()]);
  Category category =
    Category.valueOf(csvFields[CATEGORY.ordinal()]);

  books.add(new Book(title, author, pages, category));
}
```

Jackson 库可以使用类中的字段名作为 JSON 中的键，将大部分的 Java 类转换为等效的 JSON 表示。使用 Jackson 的 ObjectMapper 类，我们只需要两行代码就可以将上面的 Book 对象的 List 转换到 JSON 文件中：

```
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(outputJson, books);
```

与之类似，也可以使用 Apache Commons CSV 库简化 CSV 的解析过程。CSVParser 类可以使用 parse 方法读取一个 CSV 文件并用 withHeader 方法把标签赋给每一列：

```
List<CSVRecord> records = CSVFormat
  .DEFAULT
  .withHeader(TITLE.name(), AUTHOR.name(),
              PAGES.name(), CATEGORY.name())
  .parse(new FileReader(inputCsv))
  .getRecords();
```

我们现在不用手动敲逗号去分隔每一行，也不用为列的下标烦恼。我们可以遍历从 CSVParser 中得到的记录，按照名称读取每一列的内容：

```
for (CSVRecord record : records) {
  String title = record.get(TITLE);
  String author = record.get(AUTHOR);
  Integer pages = Integer.parseInt(record.get(PAGES));
  Category category = Category.valueOf(record.get(CATEGORY));

  books.add(new Book(title, author, pages, category));
}
```

我们不再需要手动编写有很多 bug 的代码，而是利用流行的、久经考验的开源库。这样代码会更短，看起来更像惯用的 Java，bug 更少，重复也更少。下面是整个 convertCsvToJson 函数：

```
public void convertCsvToJson(File inputCsv,
                             File outputJson) throws IOException {
  List<Book> books = new ArrayList<>();
  List<CSVRecord> records = CSVFormat
    .DEFAULT
```

```
    .withHeader(TITLE.name(), AUTHOR.name(),
                PAGES.name(), CATEGORY.name())
    .parse(new FileReader(inputCsv))
    .getRecords();

  for (CSVRecord record : records) {
    String title = record.get(TITLE);
    String author = record.get(AUTHOR);
    Integer pages = Integer.parseInt(record.get(PAGES));
    Category category = Category.valueOf(record.get(CATEGORY));

    books.add(new Book(title, author, pages, category));
  }

  ObjectMapper mapper = new ObjectMapper();
  mapper.writeValue(outputJson, books);
}
```

6.6 单一职责原则

单一职责原则（single responsibility principle，SRP）规定了每一个类、函数和变量都应该只有一个单一的目的。如果从另一个角度看的话，就是每一个类、函数和变量都应该有且只有一个改变的原因。举例来说，`convertCsvToJson` 函数就违反了单一职责原则。如果我们要用不同的方法读取 CSV 数据（例如从网络上读取而不是从磁盘读取），或者必须用不同的方式去解析 CSV 格式（例如添加了新的列），或者你必须用不同的方式去输出 JSON（例如把它写到控制台而不是写到磁盘），这一切都会成为修改 `ConvertCsvToJson` 的理由。也就是说，这样一个单独的函数具有过多的职责，每当我们修改其中的某一项，都是在冒着破坏其他全部功能的风险。

有一种改进的方法，就是把每一项职责都放到独立的函数中。首先，要把将一行 CSV 格式的数据转换为一个 Java 对象的代码拿出来放到一个单独的函数中，名为 `parseBookFromCsvRecord`：

```
public Book parseBookFromCsvRecord(CSVRecord record) {
  String title = record.get(TITLE);
  String author = record.get(AUTHOR);
  Integer pages = Integer.parseInt(record.get(PAGES));
  Category category = Category.valueOf(record.get(CATEGORY));

  return new Book(title, author, pages, category);
}
```

接下来，创建一个叫 `parseBooksFromCsvFile` 的函数去读取 CSV 文件，并用 `parseBookFromCsvRecord` 将其转换为 `Book` 对象的 `List`：

```
public List<Book> parseBooksFromCsvFile(File inputCsv)
  throws IOException {

  List<CSVRecord> records = CSVFormat
    .DEFAULT
    .withHeader(TITLE.name(), AUTHOR.name(),
                PAGES.name(), CATEGORY.name())
```

```
        .parse(new FileReader(inputCsv))
        .getRecords();

    List<Book> books = new ArrayList<>();

    for (CSVRecord record : records) {
      books.add(parseBookFromCsvRecord(record));
    }

    return books;
  }
```

最后把将 Book 对象的 List 转换为 JSON 的代码放到一个独立函数中，名为 writeBooksAsJson：

```
public void writeBooksAsJson(List<Book> books,
                             File outputJson) throws IOException {
  ObjectMapper mapper = new ObjectMapper();
  mapper.writeValue(outputJson, books);
}
```

这三个辅助函数都具有单一的职责，如果其中一个的职责发生了变化，我们就可以修改相关的函数，不用担心会影响到负责其他职责的代码。如果把这些辅助函数放到一起，可以把 convertCsvToJson 减少到短短两行：

```
public void convertCsvToJson(File inputCsv,
                             File outputJson) throws IOException {
  List<Book> books = parseBooksFromCsvFile(inputCsv);
  writeBooksAsJson(books, outputJson);
}
```

6.7 函数式编程

遵循单一职责原则会使设计出现许多短小的、简单的、独立的函数，每个函数都容易阅读、维护和测试。而且，我们还可以把这些函数中的几个组合在一起，创建具有更复杂行为的函数，这就是函数式编程的基本原理：使用函数和函数的组合作为应用程序的构建块。其中的关键就是用一种安全且容易组合的方式去设计函数。

函数式编程可以说是一个庞大的主题，Java 并不是实现函数式编程的理想语言，所以这一节只会简要介绍函数式编程背后的基本理念，并阐述如何用这些理念得到更加整洁的代码。我将要介绍的基本概念有不可变数据、高阶函数和纯函数。

6.7.1 不可变数据

看看下面的代码：

```
public class Groceries {
  public List<String> shoppingList = new ArrayList<>();

  public void fillShoppingList() {
    shoppingList.add("milk");
    shoppingList.add("eggs");
    shoppingList.add("bread");
```

```
      if (!isOnDiet()) {
        addCandy(shoppingList);
      }

      if (isXmas()) {
        addXmasFoods(shoppingList);
      }
    }
  }
```

如果我们调用 `fillShoppingList` 函数，`shoppingList` 字段中存放的值是什么？开始的时候，`shoppingList` 字段是空的，之后的值是 `["milk", "eggs", "bread"]`。接下来就不是很明显了，`addCandy` 和 `addXmasFoods` 方法获得了 `shoppingList` 的引用，所以必须阅读这些函数中的代码才能了解它们对 `shoppingList` 做了什么。实际上，由于 `shoppingList` 是 `Groceries` 类中的一个字段，该类中的任何方法都可以对它进行修改，所以我们还必须阅读 `isOnDiet` 和 `isXmas` 中的全部代码。而且，因为 `shoppingList` 是一个公共字段，可以被任何访问 `Groceries` 类的对象修改，这就意味着，除非把整个代码库都爬一遍，否则是无法确定 `shoppingList` 中的值的。如果在寻找的过程中，还发现 `Groceries` 类被用在了多线程环境中，我们根本就没办法知道 `shoppingList` 中存放的究竟是什么，因为它的值取决于线程执行的不确定顺序。

换句话说，即便在这样的一小段代码中，我们也很难推导出 `shoppingList` 的值，因为它是**可变变量**。可变变量是指向内存位置的指针，该位置也许会在不同的时间存入不同的值。一旦考虑时间因素，问题就变得很困难了，我们必须在阅读全部代码的时候，随时记着 `shoppingList` 的状态，要兼顾考虑所有可能的时间轴才能确定出它的值。随着可变变量应用范围的扩大，随着并发性的引入，可能的时间轴的数量会以指数形式增加，代码的跟踪就变成不可能了。

更好的方法是使用**不可变变量**。不可变变量只不过是固定值的一个标识符，永远不会有变化。在大多数函数式编程语言（比如 Haskell）中，默认使用的是不可变变量。只要把变量名和值关联起来，它就永远不会改变。例如，如果把 `x` 变量的值设置为 5，后面又尝试将其修改为 6，就会得到一个编译错误：

```
x = 5
x = 6 -- 编译错误!
```

在非函数式编程语言中，可变性是默认的，但是通常也提供了将变量标记为不可变的途径。举例来说，在 Java 中，可以把一个变量声明为 `final`，这样只要给它一个值，它就永远不会改变：

```
final int x = 5;
x = 6; // 编译错误!
```

如果我们用的是对象而不是原始类型，最好的实践就是将对象中的所有字段设置为不可变：

```
public class Person {
  private final String name;
  private final int age;

  public Person(String name, int age) {
    this.name = name;
```

```
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public Person withName(String newName) {
    return new Person(newName, age);
  }

  public Person withAge(int newAge) {
    return new Person(name, newAge);
  }
}
```

注意 Person 类中的所有字段都被声明为 final。另外，虽然代码中有常见的 getter 方法，但没有 setter 方法，而是用 withX 方法返回 Person 类的新实例。你很可能在以前已经多次用过这样的不可变类，例如在 Java 中，String 类就是不可变的。我们对不可变变量唯一可以做的就是用它们去计算产生新的值：

```
newValue = someComputation(oldValue);
```

在 String 类中，所有看似对 String 进行修改的方法实际上返回的都是一个新的 String：

```
String str1 = "Hello, World!";
String str2 = str1.replaceAll("l", "");

// str1仍然是"Hello, World!"
// str2是"Heo, Word!"
```

大多数语言对于常用的数据结构也提供了不可变的实现。举个例子，对于 Java 来说，Google Guava 库提供了 Set、Map 和 List 的不可变版本：

```
List<String> shoppingList =
  ImmutableList.of("milk", "eggs", "bread");
```

虽然有些问题只能通过可变变量解决，但在编写大部分代码时，仍然可以也应该使用不可变变量。一般的策略是从一个初始值开始，不直接修改它，而是一步一步地将其转换为新的**中间值**，直至得到某个需要的结果：

```
originalValue = getOriginalValue();

intermediateValue1 = computation1(originalValue);
intermediateValue2 = computation2(intermediateValue1);
intermediateValue3 = computation3(intermediateValue2);

desiredResult = finalComputation(intermediateValue1,
                                 intermediateValue2,
                                 intermediateValue3);
```

注意在上述模式中，我们从没有修改任何旧的值，而是不断地从每一次计算中生成新的中间值。可以用这种策略让 Groceries 类的推导变得更加容易：

```
public List<String> buildShoppingList() {
  List<String> basics =
    ImmutableList.of("milk", "eggs", "bread");
  List<String> candy =
    !isOnDiet() ? getCandy() : emptyList();
  List<String> xmas =
    isXmas() ? getXmasFoods() : emptyList();

  return new ImmutableList.Builder<String>()
    .addAll(basics)
    .addAll(candy)
    .addAll(xmas)
    .build();
}
```

其思路就是将每一次的食物“计算”结果存放到永远不会改变的本地的中间 List 对象中。最后，把所有的 List 连接成一个新的 List 并通过函数返回。由于每一个中间 List 对象都有一个名称，所以代码的逻辑就会变得更容易理解。并且因为一切对象均是不可变的，buildShoppingList 函数的代码逻辑现在完全是本地的，没有其他的函数、类或者线程会对结果有任何的影响。有了不可变数据，就不再需要费心考虑时间轴的问题了。

6.7.2 高阶函数

我们可以很轻松地把 buildShoppingList() 方法中的所有变量改成不可变的，因为已经提前知道要执行的“计算”的次数，所以可以把每一次计算结果都赋给一个命名的中间不可变变量，最后再把它们全部连接起来。但是如果计算执行的次数是不确定的又要怎么办呢？举例来说，在 BookParser 代码的 parseBooksFromCsv 方法中，记录的数量取决于 CSV 文件的内容：

```
public List<Book> parseBooksFromCsvFile(File inputCsv)
  throws IOException {

  List<CSVRecord> records = CSVFormat
    .DEFAULT
    .withHeader(TITLE.name(), AUTHOR.name(),
                PAGES.name(), CATEGORY.name())
    .parse(new FileReader(inputCsv))
    .getRecords();

  List<Book> books = new ArrayList<>();

  for (CSVRecord record : records) {
    books.add(parseBookFromCsvRecord(record));
  }

  return books;
}
```

既然必须进行的“计算”（即调用 parseBookFromCsvRecord）次数是无法提前预知的，那

么如何才能在没有可变变量的情况下构造出 Book 对象的 List 呢[4]？一种解决的办法就是使用**高阶函数**，这是一种能够把其他函数作为参数的函数。

Java 增加了对高阶函数的支持，比如 map、filter 和 reduce，在 Java 语言的第 8 个版本中，它已经成为了 Stream API 的一部分。通过比较把 Integer 类型的 List 中的所有偶数对象相乘的不同实现方法，可以看出它的实际应用。下面是命令式的解决方案：

```
List<Integer> numbers = Lists.newArrayList(1, 2, 3, 4, 5);
int product = 1;

for (int i = 0; i < numbers.size(); i++) {
  int number = numbers.get(i);
  if (number % 2 == 0) {
    product = product * number;
  }
}

// product的值现在是8
```

命令式的解决方案要求我们必须关注一些底层的、容易出错的细节，比如迭代、列表下标，还要维护一个可变变量去计算 product 的值。下面是同一个问题的函数式解决方案：

```
List<Integer> numbers = ImmutableList.of(1, 2, 3, 4, 5);
final int product = numbers
  .stream()
  .filter(number -> number % 2 == 0)
  .reduce((a, b) -> a * b)
  .orElse(1);

// product的值现在是8
```

函数式解决方案让我们关注一些高级的细节，比如如何找出偶数、如何让两个数字相乘，完全不需要维护任何的可变变量。我们可以使用高阶函数去掉 parseBooksFromCsv 函数中的可变因素：

```
public List<Book> parseBooksFromCsvFile(File inputCsv)
  throws IOException {

  List<CSVRecord> records = CSVFormat
    .DEFAULT
    .withHeader(TITLE.name(), AUTHOR.name(),
                PAGES.name(), CATEGORY.name())
    .parse(new FileReader(inputCsv))
    .getRecords();

  return records
    .stream()
    .map(this::parseBookFromCsvRecord)
    .collect(Collectors.toList());
}
```

注 4：在这一特定场景下，事实上 books 是可变变量也无须我们去关注，它是一个不传递给其他任何函数的变量，而且代码也很短。根据项目中的编码约定，保持这种方式不变也许也是合理的。然而，这样的代码可能会变得更长和更复杂，所以如果要重构代码或者从头开始编写类似的代码，最好就是养成在可能的情况下使用不变变量的习惯。

6.7.3 纯函数

使用不可变数据和高阶函数可使代码更容易理解和维护。但是想要利用函数式编程的最大优势，还需要使用**纯函数**。纯函数满足以下条件。

- 该函数是幂等的：给定相同的输入参数，函数总是返回精确的相同结果。
- 该函数没有**副作用**：函数不以任何方式依赖或修改外部世界的状态。副作用的例子包括改变全局变量、写入到硬盘、读取用户控制台的输入、通过网络接收数据。

纯函数唯一可以做的事情就是对输入参数进行转换并返回新的值。这不仅使纯函数的推导变得简单，也可以很容易地把它们组合起来。只要一个纯函数的返回值是另一个纯函数的有效参数，对它们的组合就肯定是安全的：

```
result = pureFunction3(pureFunction2(pureFunction1(val)));
```

有副作用的函数在组合时就更困难一些。举例来说，BookParser 中的 convertCsvToJson 函数会对文件系统进行读写，所以它不是纯的：

```
public void convertCsvToJson(File inputCsv,
                             File outputJson) throws IOException {
```

注意该函数的签名中并没有返回值（它是 void 函数），这是有副作用的函数的典型特征[5]。没有了返回值，我们就很难把这个函数和其他函数组合起来，这些函数将不得不通过文件系统或共享的可变变量进行通信，这两种方式都会比使用参数和返回值更加复杂，也更容易出错。

即便一个函数是有返回值的，它仍然可能会有副作用，这样对函数的推导和组合会更加困难。例如，要推导出 convertCsvToJson 的行为，光看其内部的代码或者它的签名是不够的，我们还必须知道外部世界的状态，比如 CSV 文件是否存在？有没有读取它的权限？如果在读取的时候有人开始覆盖文件会怎么样？ JSON 文件是不是已经存在了？有没有权限往里面写东西？如果在写入的时候，刚好有人开始写东西进去怎么办？硬盘上有没有足够的磁盘空间可以写入 JSON 文件？

不可变数据要求你要在脑海中弄清楚多条时间轴，副作用函数则要求你要在脑海中弄清楚多条时间轴和多个可能的全局状态。把几个有副作用的函数组合起来可能会引起所有时间轴和状态相互间进行交互，导致复杂度呈现指数式增长。

> 我认为可重用性的缺乏存在于面向对象语言，而不是函数式语言，因为面向对象语言的问题是它们离不开各种隐性的环境。你要一根香蕉，但得到的却是一只拿着香蕉的大猩猩和整个丛林。
>
> 如果你的代码是引用透明的，如果你有纯函数——所有的数据都来自输入的参数，输出的所有东西也没有留下什么状态——那么它的可重用性将是惊人的。
>
> ——Joe Armstrong，Erlang 之父

如果我们写的多是像纯函数一样的代码，就会发现代码变得更容易推导和重用。当然，如果想让代码有些用处，某些时候和现实之间的交互就是不可避免的，所以也无法完全

注 5：事实上，如果没有返回值，一个函数唯一的功能就是执行有副作用的操作。

摆脱副作用，最好的办法就是控制和管理这些副作用。

在像 Haskell 这样的编程语言中，只有运行时才能够执行有副作用的操作。如果我们尝试在自己的代码中直接执行有副作用的操作，就会得到一个编译错误。看看下面这句伪代码：

```
def main():
  someSideEffect()
```

如果这是 Haskell 代码，那是没法通过编译的，因为代码尝试直接去执行一个有副作用的操作。如果想要在 Haskell 中实现上述代码的功能 ，我们需要把有副作用的代码放在名为 IO 的类型中，这样代码从 main 方法返回时，Haskell 运行时才会去执行有副作用的操作[6]：

```
def main():
  return IO(someSideEffect)
```

因为 Haskell 是静态类型语言，所以 IO 类型也是函数签名的一部分，这意味着副作用操作是该语言的一等公民，我们可以对它们进行传递、组合，让编译器进行检查：

```
main:: IO ()
```

在其他大多数语言（比如 Java 语言）中，副作用在很大程度上是看不出来的。任何函数，不管它的签名是什么，都可以进行网络调用，改变全局变量，哪怕发射导弹。除了转到纯函数式语言之外，没有什么简单的解决办法。最好的方法就是把有副作用的代码尽可能地隔离到几个地方，尽最大努力保证方法签名和文档可以精确地反映所有副作用。在某些情况下，可以模仿 Haskell 的方式，把副作用操作推到应用程序的入口点，比如命令行应用程序中的 main 方法或者 Web 服务器的 HTTP 请求处理程序。

举例来说，BookParser 类并不存在读写文件的需要。这个类的目的就是解析 CSV 数据并将其转换为 JSON 数据。但事实上，这些数据是不是在硬盘上根本不重要，我们可以修改代码，把 CSV 数据作为一个 String 获取，返回的 JSON 数据也作为一个 String，完全避免和文件系统打交道：

```
public class BookParser {

  public String convertCsvToJson(String csv) throws IOException {
    List<Book> books = parseBooksFromCsvString(csv);
    return writeBooksAsJsonString(books);
  }

  public List<Book> parseBooksFromCsvString(String csv)
    throws IOException {

    List<CSVRecord> records = CSVFormat
        .DEFAULT
        .withHeader(TITLE.name(), AUTHOR.name(),
                    PAGES.name(), CATEGORY.name())
        .parse(new StringReader(csv))
        .getRecords();
```

注 6：IO 实际上是一个 monad，这是一种通用的结构，它让组合、链接和修饰变得简单，从而可以封装和拆封任意的计算（不仅仅是具有副作用的操作）。

```
    return records
        .stream()
        .map(this::parseBookFromCsvRecord)
        .collect(Collectors.toList());
  }

  public Book parseBookFromCsvRecord(CSVRecord record) {
    String title = record.get(TITLE);
    String author = record.get(AUTHOR);
    int pages = Integer.parseInt(record.get(PAGES));
    Category category = Category.valueOf(record.get(CATEGORY));

    return new Book(title, author, pages, category);
  }

  public String writeBooksAsJsonString(List<Book> books)
    throws JsonProcessingException {

    ObjectMapper mapper = new ObjectMapper();
    return mapper.writeValueAsString(books);
  }
}
```

现在 BookParser 中的每一个函数都是纯函数：每个函数都读入某些输入参数，做一些转换之后返回一个值，期间没有产生任何副作用。这样就使这些函数易于阅读、维护和重用。而 BookParser 的客户端既可以控制填入 CSV 数据的方式，也可以控制对 JSON 输出的处理，我们还可以把这段代码用在更广泛的用途中。比如下面的代码就展示了从命令行使用 BookParser 的方式：

```
public class Main {
  public static void main(String[] args) throws IOException {
    String inputCsv = args[0];
    String outputJson = args[1];

    String csv = IOUtils.toString(new FileInputStream(inputCsv));
    String json = new BookParser().convertCsvToJson(csv);

    IOUtils.write(json, new FileOutputStream(outputJson));
  }
}
```

过去所有存在于 BookParser 中的副作用（即从磁盘读写的操作），现在都被隔离到 main 方法中了。这也是这个应用的入口点，是天生进行 I/O 操作的地方。事实上，所有 BookParser 函数现在都变成了纯函数，所以也可以很轻松地编写它们的单元测试。例如下面是针对 convertCsvToJson 函数的一个简单的 JUnit 测试（阅读 7.2.1 节了解更多信息）：

```
@Test
public void testConvertCsvToJson() throws Exception {
  String csv = "Code Complete,Steve McConnell,960,nonfiction";

  String expected = "[{\"title\":\"Code Complete\"," +
                    "\"author\":\"Steve McConnell\"," +
                    "\"pages\":960," +
                    "\"category\":\"nonfiction\"}]";
```

```
    String actual = new BookParser().convertCsvToJson(csv);
    Assert.assertEquals(expected, actual);
  }
```

当我们不需要担心副作用的时候，这个单元测试就更容易编写了：不需要清理硬盘上的 CSV 或 JSON 文件；当代码并行执行的时候，测试也不存在覆盖彼此文件的可能；完成后不需要清理任何文件。

6.8　松耦合

先来看看这个 NewsFeed 类：

```
public class NewsFeed {
  List<Article> getLatestArticlesSharedByUser(User user) {
    long userId = user.data().getProfile().getDatabaseKeys().id;
    List<Article> articles = GlobalCache.get(userId);

    if (articles == null) {
      Date oneMonthAgo = new DateTime().minusDays(30).toDate();

      String query =
        "select * from articles where userId = ? AND date > ?";
      articles = parseArticles(DB.query(query, userId, oneMonthAgo));

      GlobalCache.put(userId, articles);
    }

    return articles;
  }
}
```

NewsFeed 类中的 getLatestArticlesSharedByUser 方法尝试获取特定用户在过去 30 天内分享的文章。它会先在缓存中查找，如果缓存没有命中的话就退回去调用数据库。这段代码存在许多问题，找出这些问题的好办法就是尝试编写一个单元测试（阅读 7.2.1 节，了解如何通过测试获得更好的设计）：

```
@Test
public void testGetLatestArticlesSharedByUserFromDB() {
  // 开始时为空缓存，一切都不在缓存中
  GlobalCache.reinit();
  // 开始时为空数据库
  DB.reinit();

  // 生成两篇近期的文章
  Article article1 = new Article("Recent Article 1");
  Article article2 = new Article("Recent Article 2");

  // 创建用户对象
  long userId = 5;
  User user = createMockUser(userId);

  // 把文章插入数据库
  String insertStatement =
    "insert into Articles(userId, title, date) values ?, ?, ?";
```

```
    DB.insert(insertStatement, userId,
              article1.getTitle(), new Date());
    DB.insert(insertStatement, userId,
              article2.getTitle(), new Date());

    // 保证新闻源返回两篇文章
    List<Article> actualNews =
      new NewsFeed().getLatestArticlesSharedByUser(user);
    List<Article> expectedNews =
      Arrays.asList(article1, article2);

    assertEquals(expectedNews, actualNews);
}
```

考虑一下，如果要通过测试，我们得把 NewsFeed 代码的多少内部实现细节和假设复制到 TestNewsFeed 类中。

- 测试代码必须知道要调用 GlobalCache.reinit()，这样 getLatestArticlesSharedByUser 中的缓存查询才不会让未初始化的缓存出问题。如果某一天 NewsFeed 类使用了不同的缓存策略，测试代码就会出问题。
- 测试代码隐含知道要调用 DB.reinit()，以确保数据库上线并运行，模式已经部署好，表也全都初始化为空（即不存在其他测试生成的记录）。该测试也清楚地知道数据的模式，这样就可以使用 DB.executeInsert() 插入模拟的文章。如果有一天 NewsFeed 类改变了存储数据的方式，比如用不同的数据库模式或者使用键 – 存储值去代替数据库，测试代码也将出问题。
- TestNewsFeed 类必须知道 getLatestArticlesSharedByUser 是从 User 对象抽取出用户 id，这样它才能够准确生成 User 对象，以及有相同用户 id 的文章列表。如果有一天 User 对象发生了变化，或者 NewsFeed 类使用了不同的缓存键（例如 User 类的 hashCode），测试代码将会出问题。
- 测试也需要掌握 getLatestArticlesSharedByUser 定义的“最新”文章是指最近 30 天内的文章。它要利用这一信息确保插入数据库中文章的日期会被 getLatestArticlesSharedByUser 内的查询代码挑选出来。如果有一天 NewsFeed 修改了“最新”的定义（例如最近 5 天内的文章），测试代码就会出问题。
- NewsFeed 类正使用 GlobalCache 存放用户 id 和最新文章的映射，并且假设没有其他代码在使用这一缓存。然而，缓存是全局的，如果某天代码库中其他地方的开发人员把不同类型的数据存放到缓存中，将会导致 NewsFeed 代码出现问题。

每当我们需要改变 NewsFeed 类，即便只是改变对外没有影响的内部实现细节，可能也必须对测试代码进行更新。更糟糕的是，NewsFeed 类的所有客户端都必须做出许多和测试代码一样的假设，所以我们也必须对它们进行更新。在软件领域，两个模块相互之间的依赖程度称为**耦合**。如果无论什么时候更新一个模块，都不得不频繁地更新另一个模块，这些模块就是**紧耦合**的，这通常表明代码是脆弱而且难以维护的。

整洁的代码应该遵循**依赖反转原则**：

- 高级的模块不应该依赖于低级的模块，二者都应该依赖于抽象；
- 抽象不应该依赖于细节，细节应该依赖于抽象。

NewsFeed 类就是因违反依赖反转原则，从而导致紧耦合的四种常见方式的一个例子：

- 内部实现依赖性：User 类；
- 系统依赖性：时间；
- 库依赖性：DB 类；
- 全局变量：GlobalCache 类。

6.8.1　内部实现依赖性

来看看 NewsFeed 类是如何从 User 类中提取用户 id 的：

```
long userId = user.data().getProfile().getDatabaseKeys().id;
```

一长串的方法调用和字段查询通常就是紧耦合的标志[7]。在上面的代码中，因为 NewsFeed 类深入到 User 类的内部，所以 User 类一旦发生变化（例如改变 id 字段的名称或者把用户存放在键 - 值存储而不是数据库中），就必须对 NewsFeed 进行更新。

这是违背依赖反转原则的，因为 NewsFeed 类取决于 User 类的底层实现细节，而不是对获取用户 id 的方式进行了高级抽象。这也是一个引入更多的名称可以得到更整洁的代码的情况，比如在 User 类中公开 getId() 方法：

```
public long getId() {
  return data().getProfile().getDatabaseKeys().id;
}
```

对 getId() 更高级抽象的依赖降低了耦合。而在底层，你可以按照自己的方式在 User 类中实现 getId() 方法，随时改变具体的实现，不需要改变 NewsFeed 类或其他任何客户端。

6.8.2　系统依赖性

来看看 getLatestArticlesSharedByUser 中的这行代码：

```
Date oneMonthAgo = new DateTime().minusDays(30).toDate();
```

它调用 new DataTime() 查询当前的日期和时间，所以 getLatestArticlesSharedByUser 不是幂等的：每次运行代码都会有不同的表现，对系统时钟的依赖性使得代码的测试和推导变得更加困难。

更好的设计是依赖性**注入**，即让客户端传递依赖性，而不是把它硬编码在 getLatestArticlesSharedByUser 中，将依赖关系反转过来。在可能的情况下，依赖性注入的最佳方式就是把它们作为函数参数进行传递，因为如果函数的所有依赖项在函数签名中是看得见的话，推导函数的结果也会更容易一些。举例来说，可以添加名为 since 的日期参数，并将函数重命名为 getLatestArticlesSharedByUserSince，表明它将返回特定用户自某个具体日期开始，分享的所有文章：

注 7：这一点并不适用于专门将方法调用链接在一起的代码，因为这些代码通常每次返回的都是相同的数据类型，而不是进入到类的内部。例如在对集合进行函数式转换（例如 list.filter(i → i > 5).map(i → i + 2).sum()），或对构造器类进行函数式转换（例如 CacheBuilder.newBuilder().maximum-Size(1000).expireAfterWrite(10, TimeUnit.MINUTES).build()）的时候，可以把方法调用链接在一起来实现。

```
List<Article> getLatestArticlesSharedByUserSince(User user,
                                                 Date since) {
  List<Article> articles = GlobalCache.get(user.getId());

  if (articles == null) {
    String query =
      "select * from articles where userId = ? AND date > ?";

    articles = parseArticles(DB.query(query,
                                      user.getId(),
                                      since));
    GlobalCache.put(user.getId(), articles);
  }

  return articles;
}
```

该函数对系统时钟不再有依赖性，所以它的表现是幂等的，可以更容易地理解和测试。这样的函数也更加灵活。如果我们以后遇到需要过去 60 天（而不是过去 30 天）内文章的使用场景，只要传入一个不同的 `since` 参数即可，不用非得写一个新函数。

6.8.3 库依赖性

`NewsFeed` 类使用一个叫 `DB` 的库去访问数据库：

```
String query =
  "select * from articles where userId = ? AND date > ?";
articles = parseArticles(DB.query(query, userId, oneMonthAgo));
```

`NewsFeed` 代码不应该关注或者知道文章是存放在数据库中的。它所关注的是它能够以某种方式取出满足特定条件的文章——至于底层发生什么那是别人的问题。换句话说，这一代码违背了依赖反转原则：`NewsFeed` 类依赖于底层的数据库访问细节，而不是依赖于检索文章的高级抽象。想要在 Java 中定义抽象，我们可以使用**接口**：

```
public interface ArticleStore {
  List<Article> getArticlesForUserSince(long userId, Date since);
}
```

如何才能将 `ArticleStore` 的实例注入 `NewsFeed` 类中呢？我们可以把它作为参数传给 `getLatestArticlesSharedByUserSince` 函数，但是会导致 API 冗长而混乱。根据不同的编程语言，注入依赖性的方式有很多。在 Java 和其他面向对象语言中，注入依赖性的简单方法是把它们作为构造函数的参数。Java 也提供了专门实现依赖性注入的库和框架，通常称为控制反转（Inversion of Control，IoC）容器，比如 Spring 和 Guice。在 Scala 中，我们可以通过蛋糕模式（cake pattern）注入依赖性；而在 Haskell 中，我们也许要使用函数柯里化（ function currying）或者读取单子（reader monad）模式。不论选择什么样的依赖性注入技术，我们的目的都是让依赖性成为 API 一个可见又明显的部分。

试试在 `NewsFeed` 的例子中使用构造函数注入，使该类的任何用户都清楚地知道它对获取文章数据的方式存在依赖：

```
public class NewsFeed {
  private final ArticleStore articleStore;
```

```
  public NewsFeed(ArticleStore articleStore) {
    this.articleStore = articleStore;
  }

  List<Article> getLatestArticlesSharedByUserSince(User user,
                                                   Date since) {
    List<Article> articles = GlobalCache.get(user.getId());

    if (articles == null) {
      articles =
        articleStore.getArticlesForUserSince(user.getId(), since);
      GlobalCache.put(user.getId(), articles);
    }

    return articles;
  }
}
```

上述代码将获取文章数据的实现细节和 `NewsFeed` 本身的实现细节进行了解耦。现在我们可以查询关系型数据库，传入实现了 `ArticleStore` 接口的 `DatabaseArticleStore`，或者查询文档数据库，传入 `DocumentArticleStore`，甚至传入将文章存放在内存 `HashMap` 的 `InMemoryArticleStore`，这在测试上是很有用的。抽象的强大之处在于我们可以把任何想要的改变放到 `ArticleStore` 的实现中，而不需要修改 `NewsFeed` 的代码。

现在有一个棘手的问题，我们应该把哪些库暴露为依赖项？举例来说，`BookParser` 代码使用 Jackson 库将 Java 对象转换为 JSON，并使用 Apache Commons CSV 库去解析 CSV 文件。我们也应该把这些库作为依赖项去注入吗？根据经验，我们更应该对一些有如下特征的库的具体实现进行注入抽象。

- 具有副作用。
- 在不同的环境中有不同的表现。

例如在 `BookParser` 代码中，我们不需要注入 Jackson 或 Apache Commons CSV 库，因为我们只是使用内存中的 `String`，所以不存在任何副作用；而且这些库的表现在所有环境中都是一样的。换句话说，在 `NewsFeed` 的代码中，我们应该把 `ArticleStore` 作为注入依赖项，因为它可能会通过网络 I/O 与远程数据库进行通信，而且我们也许会在不同的环境中使用不同的数据库，甚至在测试的时候使用模拟的数据库。

6.8.4 全局变量

`NewsFeed` 类中问题最大的依赖性就是使用了 `GlobalCache` 类：

```
List<Article> articles = GlobalCache.get(userId);

// ...

GlobalCache.put(userId, articles);
```

正如它的名称所表达的，`GlobalCache` 是一个**全局变量**，即可以被代码库中所有代码访问的可变状态。如果 `NewsFeed` 的用户忘记在调用 `getLatestArticlesSharedByUserSince` 之前初始化 `GlobalCache`，会发生什么呢？如果不止一个用户去初始化 `GlobalCache`，会怎

么样呢？如果 NewsFeed 被多个线程使用，又会发生什么？如果一些不相关的代码使用 GlobalCache 去存储用户不同的文章列表呢？没有使用 userId 作为键，又会发生什么？

全局变量是危险的。根据定义，它们到处都可以被访问，所以当我们使用全局变量时，实际上是对整个代码库的耦合。全局变量在不同的语言中以不同的形式出现，比如在 Java 中是 static 关键字，在 JavaScript 中是 window scope，在 Ruby 中变量名是以 $ 开头的，在 PHP 和 Python 中是 global 关键字，或者所有语言中的可变单例。只要有可能，我们就要避免使用全局变量——99% 有更好的解决方案。

如果你正在处理遗留的代码，和全局变量做斗争，你可以遵循依赖反转原则来降低这种危害。NewsFeed 类并不需要知道实现缓存的底层细节，它需要的只是缓存的高级抽象。我们先来定义一个传递缓存的接口：

```
public interface PassthroughCache<K, V> {
  V getOrElseUpdate(K key, Supplier<V> valueIfMissing);
}
```

该接口仅由一个根据特定的键返回对应的值的方法，如果不存在与该键关联的值，就将 valueIfMissing 函数生成的值存入其中并返回该值。如果我们真的愿意，可以在底层用全局变量去实现（例如 GlobalCache），但更好的做法是使用内存缓存或分布式缓存的实例（例如 memcached）。使用抽象意味着 NewsFeed 代码并不需要知道或关注它，松耦合让我们可以选择所要的实现，后续也可以安全地改变主意。

现在将缓存抽象注入 NewsFeed 的构造函数：

```
public class NewsFeed {
  private final ArticleStore articleStore;
  private final PassthroughCache<Long, List<Article>> cache;

  public NewsFeed(ArticleStore articleStore,
                  PassthroughCache<Long, List<Article>> cache) {
    this.articleStore = articleStore;
    this.cache = cache;
  }
  List<Article> getLatestArticlesSharedByUserSince(User user,
                                                   Date since) {
    return cache.getOrElseUpdate(
      user.getId(),
      () -> articleStore.getArticlesForUserSince(user.getId(),
                                                 since));
  }
}
```

现在已经将所有的依赖性都倒置到 NewsFeed 代码中，降低了耦合，使得代码更容易维护和测试。下面是之前的单元测试的更新版本：

```
@Test
  public void testGetLatestArticlesSharedByUserSince() {
    List<Article> expected = Arrays.asList(
      new Article("Article 1"), new Article("Article 2"));
    NewsFeed newsFeed =
      new NewsFeed(new MockArticleStore(expected),
                   new AlwaysEmptyCache());

    User user = createMockUser(5);
```

```
    Date since = new Date();
    List<Article> actual =
      newsFeed.getLatestArticlesSharedByUserSince(user, since);

    assertEquals(expected, actual);
  }
```

这段测试代码很容易理解，也不太可能因为 NewsFeed 类内部的变化而出问题，并且还可以安全地并行运行多个这样的测试。

6.9 高内聚

看看乔姆斯基论文中的一句话：无色的绿色思想愤怒地沉睡（Colorless green ideas sleep furiously）。乔姆斯基用这句话作为例子，表达了一个语法正确但完全没有意义的句子。这些单词是毫不相关的，形成了没有什么逻辑的句子。

现在，考虑下面这个类：

```
public class Util {
  void generateReport() { /* ... */ }
  void connectToDb(String user, String pass) { /* ... */ }
  void fireTheMissles() { /* ... */ }
}
```

这也是一个技术上正确，但完全没有意义的类。这些方法是毫不相关的，产生了一个**低内聚**的类。

> 内聚（cohesion）一词和“附着力”（adhesion）一词有相同的词根。这是一个表示黏性的词，当什么东西附着在其他东西上时（换句话说，它是有黏性的），它是一种单面的、外部的东西：这种东西（比如胶水）能把一种东西粘到另一种东西上。换句话说，有黏着力的东西本身就会彼此粘在一起，因为它们就是这样的东西，或者因为它们可以很好地结合起来。厚布胶带可以粘住东西是因为它们是有黏性的，不是因为它们必须让任何东西都和它们一样。但是当你把两块黏土放到一起时，精心加工匹配，有时看起来就像凝聚在一起，是因为它们是精确地匹配在一起的。
>
> ——Glenn VanderBurg，Livingsocial 技术总监

类名中的单词“util”是低内聚的典型特征，名称中带有 util 的类通常是不知道要放哪里的不相关函数的大杂烩。然而，低内聚并不总像 Util 类那么明显。来看一个更加实际的例子：

```
public interface HttpClient {
  byte[] sendRequest(String url,
                     Map<String, String> headers,
                     byte[] body);
  Document getXml(String url);
  int postOnSeparateThread(String url,
                           String body,
                           ExecutorService executor);
  void setHeader(String headerName, String headerValue);
  boolean statusCode();
}
```

与 `Util` 类中的其他方法相比，伪 `HttpClient` 接口中方法的关系要更紧密一些，因为这些方法全部都与 HTTP 请求的发送有关。然而，这些方法也是低内聚的，因为它们是在许多不同的抽象层次上操作的：

- `sendRequest` 把字节数组用在请求和响应的 body 上，`postOnSeparateThread` 把 `String` 用于请求的 body 而没有返回响应的 body（它只返回一个状态代码），`getXml` 没有请求 body，它返回一个 XML `Document` 给响应的 body。
- `postOnSeparateThread` 负责底层的线程处理细节（通过 `ExecutorService`），但其他方法都不会这么做。
- `setHeader` 和 `statusCode` 方法暗示了 `HttpClient` 是可变的，可以存储下一个请求或者前一个响应的状态。这些方法如何与其他方法交互并不清楚，特别是 `sendRequest`，该方法把一个 HTTP headers 的 `Map` 对象作为参数之一。

整洁的代码应该有**高内聚**：所有的变量和方法都应该是有关联的，一切都应该在同一抽象层次上操作，每一部分都应该很好地相互配合。例如，下面用更加内聚的方法实现 `HttpClient` 接口：

```
public interface HttpClient {
  HttpResponse sendRequest(HttpRequest request);
}

public interface HttpRequest {
  URL getUrl();
  Map<String, String> getHeaders();
  byte[] getBody();
}

public interface HttpResponse {
  Map<String, String> getHeaders();
  byte[] getBody();
}
```

新的 `HttpClient` API 只有一个发送请求的单独的方法，其他所有的逻辑都由其他类去处理。例如，HTTP 头部、URL 和 body 的细节由 `HttpRequest` 和 `HttpResponse` 处理。如果我们不只手动设置 HTTP 头部和处理请求 body 的字节数组，而是要处理更高级别的请求，可以创建一个 `HttpRequestBuilder` 类：

```
public class HttpRequestBuilder {
  public HttpRequest postJson(String url,
                              String json) throws Exception {
    return new BasicHttpRequest(
      url,
      ImmutableMap.of("Method", "POST",
                      "Content-Type", "application/json"),
      json.getBytes("UTF-8"));
  }
}
```

如果要用更高级的方式去处理响应的 body 而不使用字节数组，我们可以创建一个 `HttpResponseParser` 类：

```
public class HttpResponseParser {
  public Document asXml(HttpResponse response) {
    return DocumentBuilder.parse(
      new ByteArrayInputStream(response.getBody()));
  }
}
```

任何有关线程处理的任务都可以由 `HttpClient` 实现去处理：

```
public class ThreadedHttpClient implements HttpClient {
  private final ExecutorService executor;

  public ThreadedHttpClient(ExecutorService executor) {
    this.executor = executor;
  }

  public HttpResponse sendRequest(HttpRequest request) {
    try {
      return executor.submit(() -> doSend(request)).get();
    } catch (Exception e) {
      throw new HttpClientException(e);
    }
  }
}
```

这里并没有使用一个较大的 `HttpClient` 类去处理许多不相干的任务，而是使用几个较小的类，每个类都处理几个高度相关的任务。我们从一个庞然大物转到若干专注、高度内聚的类，这是实现整洁代码的标准模式。

6.10 注释

> 不要为糟糕的代码注释——重新写吧。
>
> ——Brian W. Kernighan、P. J. Plauger，《编程格调》

我有意把介绍注释的内容放到比较靠后的位置，因为代码本身应该告诉你需要知道的几乎一切。如果代码没有做到，在你费劲地编写任何注释之前，应该先对代码进行改进。

现在 `BookParser` 代码看起来已经相当整洁，我们可以添加一条注释，对代码中无法体现的内容进行解释，比如最初为什么会有这段代码、我们对输入或输出所做的所有假设以及一些例子。因为这是 Java 代码，所以可以用 JavaDoc 来设置注释的格式：

```
/**
 * 将CSV格式的图书数据转换为JSON格式。
 * CSV文件应该使用RFC4180格式，每行包含4列：
 * author、title、pages、category。pages必须是整数。
 * category必须是“fiction”或者“nonfiction”。
 *
 * 例如，对于如下的CSV文件：
 *
 * George R.R. Martin, Game of Thrones, 864, fiction
 * Tor Norretranders, The User Illusion, 480, nonfiction
 *
 * 可以得到如下的JSON格式：
```

```
 *
 * [
 *   {
 *     "author": "George R.R. Martin",
 *     "title": "Game of Thrones",
 *     "pages": 864,
 *     "category": "fiction"
 *   },
 *   {
 *     "author": "Tor Norretranders",
 *     "title": "The User Illusion",
 *     "pages": 480,
 *     "category": "nonfiction"
 *   }
* ]
 *
 * @param csv 包含CSV格式的图书数据的字符串。
 * @return 包含图书数据的JSON表示的字符串。
 * @throws IOException 如果该CSV文件没有有效格式。
 */
public String convertCsvToJson(String csv) throws IOException {
```

注释是文档的一部分，这一点将在 7.2.4 节详细讨论。

6.11 重构

本章对 `BookParser` 代码进行了增量式地改进，每次只对内部实现细节做一点小改变，这就是所谓的**重构**。重构是改变代码结构而没有改变其外部行为的过程，这是一种只影响软件“非功能”方面的编码任务：从外部看，代码实现的功能并没有变化；但从内部看，我们已经改进了它的设计。

合理的重构是必不可少的，因为我们不可能一开始就能正确地设计。和论文的初稿一样，代码的初稿也会是凌乱、不完整、需要重写的。虽然我有意把 `BookParser` 的例子写得很难看，但任何代码实现的第一个版本总是存在问题的。随着继续编写代码，我们会更好地理解问题，而重构的本质就是回到代码中，根据这种新的理解去改进它。

> 编程语言是对程序的思考，而不是表达你已经思考过的程序。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

不管是编写代码还是写论文，都是一个迭代的过程。我们可能要经过许多份草稿才能写成一篇出色的论文，也可能要经过许多份草稿才能写出出色的代码。这意味着重构并不是一个单独的任务——不是只在项目的“清理阶段”（这样的阶段永远不会到来）才做的事情。重构是编写软件非常核心的部分，它应该是持续不断进行着的。

> 我是“先做纯粹重构，再做纯粹扩展”的忠实信徒。我们要对代码进行整理，直到可以轻易地新增下一个功能，但它在行为上不会有任何改变。之后就可以再新增下一个功能，接着再重复整个过程。
>
> ——Deam Thompson，Transarc 公司、Premier Health Exchange、
> Peak Strategy 和 Mspoke 联合创始人

William Zinser 在《写作法宝》中写道："修改是优秀作品的精髓。"我也想提出建议，重构是出色编程的精髓[8]。

6.12 小结

只要你采纳了本章介绍的这些原则，你会发现编写整洁的代码所花的时间和编写难看的代码所花的时间是差不多的。但是，阅读整洁的代码比阅读难看的代码要少花一个数量级的时间，而更新整洁的代码比更新难看的代码要少花两个数量级的时间。由于阅读、修改代码和编写新代码所花时间的比例大概是 50∶1，所以不费脑筋就可以想到：我们永远都该编写整洁的代码。

难看的代码会想方设法出现在每一个项目中，因为围绕着代码的产品、人以及生态系统都会发生变化，我们在过去做的决定可能无法满足未来的需求。所以，我们不仅在开始时要编写整洁的代码，还必须不断对代码进行重构，让它适应新的需求。创业公司的一切，包括代码，更多是进化而来，而不是提前设计的结果，而重构正是其中的一个例子。

如果没有保持整洁（如果代码没有进化），最终我们会在技术上欠下债来。就像真正的债务一样，偿还的周期越长，累积的利息就越多。技术债务在生产效率上的成本大家都知道了，但它也是一种人力成本。

> 质量并不纯粹是经济的因素，人们也需要做他们为之自豪的工作。
>
> ——Kent Beck 和 Cynthia Andres，《解析极限编程》

技术债务也会让人感到沮丧，不妨看看本章开头原来的 `BookParser` 代码：你的大脑会拒绝看它。想象一下，如果你每天的工作就是和这样的代码打交道，给你足够多的薪水也许可以说服你去做这样的工作，但再多的钱也不会让你对此感到高兴。请记住，创业与人是密不可分的，技术债务的真正成本不在于它会导致更多的 bug 和错过最后期限，而是会让人痛苦。

编程在很大程度上就是一门手艺。你会因为选择了正确的工具、努力工作并制作出精美的东西而获得深深的成就感。它的美不仅源于对用户而言漂亮的外观，还源于其精工细作的内部运作方式。优雅的解决方案会让程序员愉悦，丑陋的拼凑则会让程序员悲伤不已。而悲伤的程序员生产效率更低、表现欠佳，最终将会离开公司。

更糟糕的是，技术债务会引发技术债务。只要你容忍一小点难看的代码进入到系统中而不去解决，就可能有一大批难看的代码随之而来，直到整个结构开始崩溃。这就是典型的破窗寓言。

> 在城市中，有些建筑是漂亮而干净的，而有些建筑则破败不堪，为什么呢？犯罪和城市衰退领域的研究人员发现了一种引人关注的触发机制，这是一种非常快就能把一个干净、完好、有人居住的建筑变成破败、被人抛弃的建筑的机制。
>
> 一个破碎的窗户。

注 8：《重构：改善既有代码的设计》对重构提供了很好的指导。

> 一个破碎的窗户，如果长期以来没有修好，就会给建筑中的居民逐渐带来一种被抛弃的感觉——感觉管理者并没有在意这栋建筑。这样会再有一个窗户被损坏。人们也开始乱丢东西、涂鸦，建筑被严重损坏。很快，这栋建筑的损坏程度使得其主人都不想再维修它了，放弃的意愿就成了现实。
>
> ——Andrew Hunt 和 David Thomas，《程序员修炼之道》

破窗理论在代码上的应用和在建筑上的应用一样。如果我们的代码库到处都是丑陋的修改和乱七八糟的代码，每一个新到的开发人员都在添乱而不是将它理顺。随着难看的代码越来越多，把代码理顺就会变得越来越难，问题只会加速发展。我们要尽快地把坏掉的窗子修好，坚持编写整洁的代码，并重写得更整洁。

第 7 章

可扩展性

7.1 创业的扩展

本章会讨论在创业中需要考虑的两种可扩展性：第一种是对编码的实践过程进行扩展，以应对更多的开发人员、代码和复杂性的需求；第二种是扩展代码的性能，以应对更多的用户、流量和数据的需求。

创业的扩展和手动汽车换挡有点类似。太早扩展就像在汽车低速行驶时切换到高速挡：齿轮会越转越慢，可能还会彻底熄火；太晚扩展就像在低速挡上把油门一脚踩到底：发动机会承受巨大的压力，出现红灯告警，如果持续的时间太长，发动机会过热，根本不会达到最高速度。所以，为了能平缓前进，我们必须在合适的时间进行扩展和换挡。

有一件最重要的事情要先弄清楚：可扩展性不是一个布尔属性。我们不能说一种实践方法或者一个系统是可扩展的还是不可扩展的。我们最多能说的就是，在某些条件下，它在某些维度上可以扩展到什么程度。适用于 10 人小公司的可扩展性实践并不适用于千人的大公司；同样，将数据库扩展到每秒一百次查询和能够处理以吉字节为单位的数据的实践方式也无法适用于每秒一万次查询和以拍字节为单位的数据库。为了更快地前进，我们需要切换到完全不同的挡位上。

本章大部分重点都放在编码实践的扩展上，因为这一点在公司的早期就会非常重要，每一个创业公司概莫能外。它们就像汽车的低速挡：我们必须通过这些档位才能切换到高档位，比如性能的扩展。通常来说，只有在公司发展到后期，对用户有强大的吸引力时，性能问题才会对公司有所影响。

7.2 编码实践的扩展

编程可能会让人提心吊胆，技术债务若被低估，其中一个代价就是对开发人员心理的影响。现在有无数程序员害怕他们的工作，也许你就是其中的一员。

凌晨三点，你拿到一份 bug 报告，之后就开始查找问题。你要把它从一堆错综杂乱的 `if` 语句、`for` 循环、全局变量、短变量名和混乱的模式中把错误挑出来。那里既没有文档，也没有测试，最初写代码的开发人员早已不在公司工作。你弄不清楚那些代码究竟是干什么的，不知道它被用到哪些地方，你感到害怕。

你对上次修复一个 bug 所花的时间仍心有余悸，不料这次又多暴露三个—— 一个“微不足道”的修改就花了你两个月的时间；一次小小的性能调试就让整个系统都停了下来，把同事惹得着急上火。你对项目的时间估算开始大量地增加，发现自己动辄就会嚷嚷“这太昂贵了”或“这不可能”，你心存恐惧，害怕每次都得修改代码。

在创业公司中，一切都在不断变化。如果你已经到了害怕修改代码的地步，就意味着你需要在编码实践上进行扩展，以适应增长的需要。应对不断增长的代码库和开发团队的最重要的四个编码实践是：

- 自动化测试；
- 代码分离；
- 代码评审；
- 文档。

7.2.1 自动化测试

自动化测试会给你做出修改的自信。虽然在你的世界里，其他地方都充满了恐惧和不确定因素，但自动化测试总是作为一种稳定而平静的存在伴随着你。它们是可靠的朋友，是只要你需要，哪怕凌晨三点也会陪伴你左右的朋友。它们是你代码的守护者。带着我对 George R.R. Martin 的深深歉意，我想说，它们是编程领域的守夜者。

> 长夜将至，我从今开始守望，直至删除方休。我不允许空指针异常，不允许差一错误，不接受无限循环。我不部署到生产服务器，不争荣宠。我将以断言为生死，我是黑暗中的模拟对象，我是 CI 服务器的守卫，亦是守护程序员王国的坚实后盾。我将生命与荣耀献给自动化工具，今夜如此，夜夜亦然。

自动化测试具备了迭代式的代码测试循环的优势，这样我们每次修改仍在运行中的东西时可以充满信心，不必在脑海中记着整个程序的状态，不必担心弄坏其他人的代码，也不必一次又一次重复同样无聊、容易出错的手动测试。我们只需要运行一条测试命令，就能够快速得到正常与否的反馈。

1. 自动化测试入门

如果你对自动化测试还不太熟悉，这里先简单介绍一下。现在假设你要写个函数对一个句子中的单词进行反转：

```
reverseWordsInSentence("startups are great");
// 返回："sputrats era taerg"
```

下面先尝试用 Java 实现：

```
public class TextReverse {
  public static String reverseWordsInSentence(String sentence) {
    StringBuilder out = new StringBuilder();
    String[] words = sentence.split(" ");
```

```
    for (int i = 0; i < words.length; i++) {
      String word = words[i];
      StringBuilder reversed = new StringBuilder(word).reverse();
      out.append(reversed);
      out.append(" ");
    }

    return out.toString();
  }
}
```

我们怎么知道这段代码正不正常呢？没错，我们可以一行行盯着代码看一段时间，判断出它是可以工作的；或者也可以在 UI 中四处点一下，看看结果对不对，手动做些测试。TextReverse 这个例子是不存在 UI 的，所以可以加一个 main 方法到代码中，把结果输出到控制台：

```
public static void main(String[] args) {
  System.out.println(reverseWordsInSentence("startups are great"));
}
```

运行这段代码，将会输出如下结果：

```
sputrats era taerg
```

乍一看似乎没有什么问题，任务已经完成了，是吗？其实不完全是。如果我们不通过目测检查，不通过 main 方法，而是想要写代码去检查这些代码，情况又如何呢？为此，我们可以使用测试框架。每一种编程语言都有一些测试框架，比如 Java 的 JUnit。大部分测试框架都是将测试代码放到单独的类中，这样就不会和产品代码混在一起。下面是用 JUnit 为 TextReverse 所写的一个测试类：

```
public class TestTextReverse {
  @Test
  public void testReverseThreeNormalWords() {
    String expected = "sputrats era taerg";
    String actual = reverseWordsInSentence("startups are great");
    assertEquals(expected, actual);
  }
}
```

在 JUnit 中，我们可以用 @Test 注释对包含测试的方法进行标记，利用 assert 函数，若代码不符合某些条件则让测试失败。下面就是运行这个测试之后在控制台中看到的结果：

```
JUnit version 4.11
Time: 0.068

There was 1 failure:
1) testReverseThreeNormalWords
   (com.hello.startup.reverse.TestTextReverse)

org.junit.ComparisonFailure:

expected:<[sputrats era taerg]> but was:<[sputrats era taerg ]>
```

```
at org.junit.Assert.assertEquals(Assert.java:115)
at org.junit.Assert.assertEquals(Assert.java:144)
at com.hello.startup.reverse.TestTextReverse.
     testReverseThreeNormalWords
     (TestTextReverse.java:14)

FAILURES!!!
Tests run: 1,  Failures: 1
```

看起来测试失败了。从跟踪栈的信息中，可以看到 assertEquals 调用失败了。查看错误输出，可以看到原因所在：它期待的结果是"sputrats era taerg"，但得到的却是"sputrats era taerg "（注意最后的空格）。问题在于手动对 List 进行循环并把 String 连接起来是很容易出错的，上一章 BookParser 的例子正是出现了这一问题，解决的方法就是使用高阶函数去处理循环（通过 map）和连接（通过 collect）：

```
public static String reverseWordsInSentence(String sentence) {
  return Arrays
    .stream(sentence.split(" "))
    .map(word -> new StringBuilder(word).reverse())
    .collect(Collectors.joining(" "));
}
```

如果对新的代码重新运行测试，可以看到如下结果：

```
JUnit version 4.11
..
Time: 0.061

OK (1 tests)
```

太好了，测试通过，问题解决了。但是请等等，如果单词之间不止一个空格，情况又怎么样呢？我们可以添加另一个测试看看结果：

```
@Test
public void testReverseWordsWithTabsAndLeadingWhitespace() {
  String expected = "sputrats era taerg";
  String actual = reverseWordsInSentence("   startups are\tgreat");
  assertEquals(expected, actual);
}
```

重新运行测试，将会看到：

```
JUnit version 4.11
Time: 0.068

There was 1 failure:
1) testReverseWordsWithTabsAndLeadingWhitespace
   (com.hello.startup.reverse.TestTextReverse)

org.junit.ComparisonFailure:

expected:<[sputrats era taerg]> but was:<[   sputrats taerg era]>

at org.junit.Assert.assertEquals(Assert.java:115)
at org.junit.Assert.assertEquals(Assert.java:144)
```

```
at com.hello.startup.reverse.TestTextReverse.
     testReverseWordsWithTabsAndLeadingWhitespace
     (TestTextReverse.java:23)

FAILURES!!!
Tests run: 2,  Failures: 1
```

噢，出现了另一个 bug。这一次是因为这段代码没有正确处理空白，比如制表符和首尾的空格。下面是一个修改了的版本：

```
public static String reverseWordsInSentence(String sentence) {
  return Arrays
    .stream(sentence.trim().split("\\s+"))
    .map(word -> new StringBuilder(word).reverse())
    .collect(Collectors.joining(" "));
}
```

现在重新运行一次测试，可以看到两个测试都通过了：

```
JUnit version 4.11
..
Time: 0.063

OK (2 tests)
```

像 `reverseWordsInSentence` 这样简单、小型的函数，手动检查代码和输出结果都会错过好几个 bug。随着代码库的增长，手动测试也变得更加没有效率。即便在小型创业公司，大部分产品也会因为有太多的使用场景和特殊情况而无法完全进行手动测试。除了上述两个测试之外，我们还要加入一些测试，对 `TextReverse` 在所有正常和特殊情况下的运行情况进行检查，包括：空字符串、单个单词、短字符串、长字符串、空白字符串和带有新行及回车的字符串。如果每种情况都是手动检查的话，花的时间将会非常多；但如果使用自动化测试，只需要零点几秒就可以完成（上面的测试花了 0.063 秒）。

自动化测试的速度非常快，足以在每次修改之后都返回结果。我们也应该在每一次签入代码后运行一下测试，作为构建过程的一个步骤（阅读 8.3 节了解更多信息）。这样一来，不管谁在使用这些代码抑或发生了其他哪些变化，我们都可以保证代码在完成编写之后还能够持续运行数月或数年。在开发、构建期间使用、运行测试，可以实现快速修改，并自信不会出现问题。这里的关键词是**自信**：因为测试通过并不能保证代码没有 bug。没有哪种形式的测试可以保证代码是没有 bug 的，这只不过是一个概率问题。我们可以通过学习编写高质量的测试，提高代码没有 bug 的可能性。

判断测试质量的一个方法就是计算自动化测试运行的时候，有百分之多少的生产代码被执行了。这一衡量尺度称为**代码覆盖**，大部分的编程语言都提供了可以自动计算代码覆盖百分比的工具，甚至还能显示出哪部分代码被覆盖了，哪部分代码没有被覆盖。例如，在 Java 中可以用 JaCoCo，如图 7-1 所示。

绿颜色的代码行是被自动化测试执行的，黄颜色的代码行是只有某些分支语句被执行的 `if` 语句，红颜色的代码行是完全没有执行的。如果我们的测试只执行了 20% 的代码，不可能信心满满地认为其他 80% 的代码是没有 bug 的；但反过来说，如果代码中 80% 的代码路径被执行而没有发现 bug，整个代码没有 bug 的可能性就要高得多。

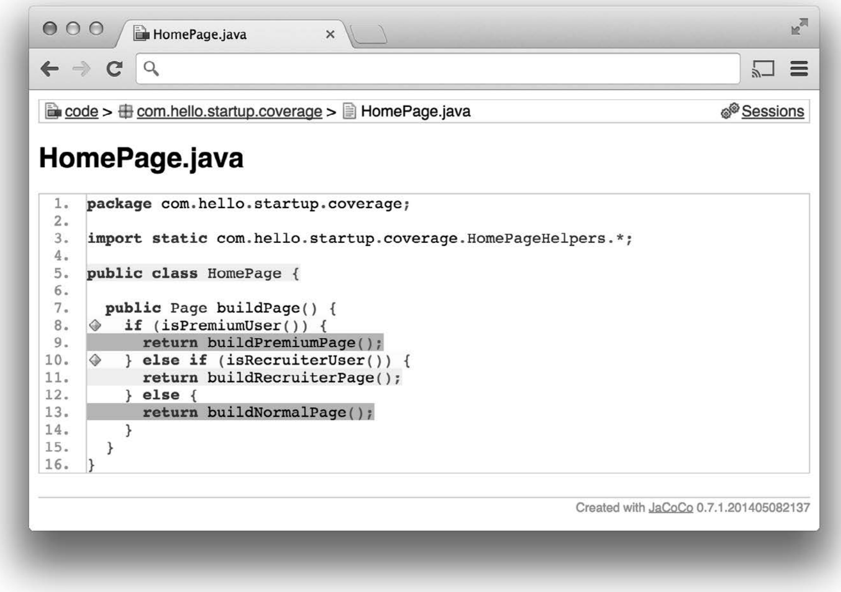


图 7-1：JaCoCo 代码覆盖

2. 自动化测试的类型

自动化测试有许多种类型，包括单元测试、集成测试、冒烟测试、验收测试和性能测试。大部分现实的应用程序都需要其中若干种类型的测试，因为每一种测试都有不同的用途，可以捕捉不同类型的 bug。我们来详细看看各种测试类型。

单元测试

单元测试验证的是单独的一小个代码单元的功能。单元（unit）并没有明确的定义，但通常都是一个单独的函数，最多也就是一个类。例如，TextReverse 代码的测试实际上都是对 `reverseWordsInSentence` 的函数进行单元测试。

单元测试是编码周期的一部分：做修改、运行测试、做修改、再运行测试。每次测试通过，你都得到了没有任何问题的反馈。这种反馈循环需要很迅速，只要花几秒钟就能够运行整个单元测试套件。因此，单元测试通常都不允许有任何副作用存在。也就是说，没有读取或写入磁盘、没有网络调用、没有数据库调用、没有访问全局变量，与这些依赖项通信需要花太长的时间，如果我们测试的单元存在这样的依赖项，就应该用一个测试替身（test double）来代替，后文将做详细讨论。

单元测试应该是第一道防线，在开发期间要不断使用，编写出不断增加的代码。单元测试是快速而可靠的，因为它们没有外部依赖项。它们能给你带来自信，让你知道应用程序的各个小模块在组合之前就是可以正常工作的。

集成测试

每个单元能各自正确运行，并不能保证多个单元组合到一起也能正确运行，所以要引入集成测试。集成测试涵盖的范围非常广泛，从测试若干个类或模块的交互，到测试

整个子系统如何一起工作（比如验证后端服务器是否可以正确使用真正的数据库）。和单元测试不同，集成测试允许产生副作用，并对外部环境存在依赖。我们仍然应该使用测试替身去代替所有和该测试没有直接关系的依赖，但如果测试的是两个子系统如何交互，就需要对这两个系统进行部署，让它们相互对话。

部署真实依赖项的缺点是集成测试要花更长的时间运行，所以不能在开发环境中经常使用。优点是即便这些测试只是针对签入之后已构建的版本，也仍然可以捕捉到许多被单元测试放过的错误，比如子系统之间的 API 不兼容。

验收测试

单元测试和集成测试侧重从开发人员的角度去验证代码的行为，回答“代码是否正确运行”的问题。验收测试则是从客户的角度验证产品的行为，回答“代码是否正确解决问题”的问题。如果创业公司编写出的能完美通过所有单元测试和集成测试的正确代码，实际上却不是客户想要的，这其实就是最大的浪费。

典型的验收测试描绘了用户的行为以及你的产品如何对其做出反应。举例来说，我们也许会用 Selenium 自动在网页浏览器中点击，检查当用户点击“like”按钮时，like 的数量是否会增 1。集成测试验证的只是几个独立的子系统之间的交互，但验收测试则是一种**端到端的测试**，验证技术栈的每一部分是否正确地结合在一起，形成可用的产品。

端到端测试的缺点是我们需要部署所有的子系统，所以测试编写起来可能会很复杂，执行速度也比较慢。端到端测试的好处是通过让代码在类生产环境中运行，少数几个测试就可以执行大量的产品代码，暴露出其他测试无法暴露的问题，比如部署、配置、服务通信和 UI 中的 bug。

性能测试

大部分的单元、集成和验收测试验证的是系统在理想条件下的正确性，即在单用户、低系统负载和无故障条件下的表现。现实情况会更混乱一些，我们的应用也许必须应对数以千计的并发用户，服务器也许在 CPU、内存和带宽等方面会受到限制，系统也将不得不面对各种各样的错误，比如缓慢或无响应的服务、服务器宕机、硬盘故障或网络问题。

性能测试的目标是验证系统在面对繁重的负载和故障时的稳定性和响应能力。性能测试涵盖了从单元测试到端对端测试的全部范围。例如，我们可以对一个排序函数进行压力测试，找出哪种算法可以得到最佳的性能；可以对搜索服务执行性能测试，测量它在处理每秒 1000 次查询时的延迟；也可以运行端到端的性能测试，用大量请求去轰炸前端服务器，看看**红线**在哪——在哪个点服务器会开始丢弃请求？是因为服务器自身资源不足还是下游（比如数据库）的问题？ 7.3 节将更深入地讨论性能问题。

3. 测试替身

当我们编写自动化测试，特别是单元测试和集成测试时，通常要独立地对每一个单元或组件进行测试。如果遇到了错误，就可以确认该错误是由单元或组件内部引起的，而非存在于它的依赖项中。为了做到这一点，我们可以在测试时用**测试替身**去代替真正的依赖项，有点像拍动作电影时，代替演员出场的特技替身。测试替身针对测试实现了便捷和快速的依赖项接口，测试替身有几种类型，包括伪造、桩和模拟，尽管很多人会在所

有场合中都只用模拟这个术语[1]。举个例子，来看看 NewsFeed 这个类：

```
public class NewsFeed {
  private final ArticleStore articleStore;

  public NewsFeed(ArticleStore articleStore) {
    this.articleStore = articleStore;
  }

  public List<Article> getLatestArticlesForUser(long userId) {
    List<Article> rawArticles = articleStore.get(userId);
    return sortAndFilter(rawArticles);
  }
}
```

函数 getLatestArticlesForUser 从分布式的键 – 值存储中获取文章，正如 ArticleStore 接口所定义的：

```
public interface ArticleStore {
  List<Article> get(long userId);
  void put(long userId, List<Article> articles);
}
```

如果我们想对 getLatestArticlesForUser 函数进行单元测试，搭建一个真正的分布式键 – 值存储系统会花很长时间。所以代替的做法是，创建一个 ArticleStore 接口的测试替身，底层使用的是内存中的 ConcurrentHashMap：

```
public class InMemoryArticleStore implements ArticleStore {
  private final Map<Long, List<Article>> store =
    new ConcurrentHashMap<>();

  public List<Article> get(long userId) {
    return store.get(userId);
  }

  public void put(long userId, List<Article> articles) {
    store.put(userId, articles);
  }
}
```

可以使用这一测试替身为 getLatestArticlesForUser 编写一个单元测试：

```
public class TestNewsFeed {
  @Test
  public void testGetLatestArticles() {
    long userId = 5;

    ArticleStore articleStore = new InMemoryArticleStore();
    articleStore.put(userId, createFakeArticles());

    NewsFeed newsFeed = new NewsFeed(articleStore);
    List<Article> actualArticles =
      newsFeed.getLatestArticlesForUser(userId);
```

注 1：读者可阅读 *xUnit Test Patterns: Refactoring Test Code* 了解其正式的定义。

```
    // .. 验证actualArticles包含模拟文章
  }
}
```

InMemoryArticleStore 不仅足以快速进行单元测试，还可以让我们控制它的行为（例如在测试的时候返回特定的值），这样我们就可以准确地验证 getLatestArticlesForUser 对所有情况的处理。从中可以看到，如果我们所测试的代码暴露了它的所有依赖项并且依赖的是高级抽象，那么使用测试替身会更加简单。读者可以阅读 6.8 节了解更多信息。

4. 测试驱动开发（TDD）

要为代码编写测试，我们要先回过头来问自己几个重要的问题：我要如何组织代码的结构才能对它进行测试？我的代码有什么依赖项？常见的使用场景是什么？会遇到什么特殊的情况？

如果发现自己的代码很难测试，就意味着它也存在其他需要重构的原因。假如代码使用了大量的可变状态并且有许多副作用，它就不只是难以测试，重用和推导代码的结果也会变得困难（阅读 6.7 节了解更多信息）；如果因为代码与其依赖项有许多复杂的交互而导致难以测试，代码可能也会由于耦合过紧而难以修改（阅读 6.8 节了解更多信息）；如果代码因为使用场景过多，测试时难以覆盖，也表明了代码的功能太多，需要进行分解（阅读 6.6 节了解更多信息）。

换句话说，测试不仅可以帮助我们编写代码，还提供了一种反馈，可以让我们获得更加出色的设计。如果我们在编写实现代码**之前**就编写测试代码，便可以从这种反馈中获得最大的好处，这就是所谓的**测试驱动开发**（Test Driven Development，TDD）。TDD 的实现过程是：

(1) 为新的功能添加测试；
(2) 运行所有测试，新的测试应该会失败，但其他所有测试应该通过；
(3) 实现这个新功能；
(4) 运行测试，现在所有测试均应该通过；
(5) 重构代码，直至拥有整洁的设计。

来从头到尾看一个使用 TDD 的例子。假设我们需要编写一个函数去读入文件，计算文件中每个单词出现的次数（但是要跳过一些**停用词**，比如 to、the 和 and），然后把单词按照出现次数从多到少的顺序进行输出。例如，如果有一个名为 four-words.txt 的文件，其内容如下：

```
Hello! Hello startup people! Hello startup world!
```

该函数应该输出如下结果：

```
hello (3)
startup (2)
people (1)
world (1)
```

下面是对函数签名的第一次推测：

```
public class WordCount {
  public void printWordCounts(File file) {
```

```
  }
}
```

因为要尝试进行 TDD，所以在填入具体的实现之前，先试着想出一些测试用例：

```
public class TestWordCount {
  @Test
  public void testPrintWordsCountOnFourWords() {
    WordCount wordCount = new WordCount();
    wordCount.printWordCounts(new File("four-words.txt"));
    // 噢，要如何检查结果呢？
  }
}
```

当我们尝试编写测试用例时，问题就很明显了：如果该函数只是将结果输出到 stdout，它是不存在返回值的。这不仅仅使其变得难以测试，也是一个糟糕设计的信号，因为我们无法在其他任务中重用该函数。例如，如果我们要在网页上显示单词的统计结果或者把单词的统计结果保存到文件中，就必须重写 printWordCounts。幸运的是，通过使用 TDD，我们很早就可以捕捉到这一问题，所以只需要重写函数的签名即可：

```
public class WordCount {
  public Map<String, Integer> calculateWordCounts(File file) {
    return null;
  }
}
```

该函数现在叫 calculateWordCounts，它不再将结果输出到 stdout，而是返回单词统计结果的 Map 对象。我们仍然可以先不做具体的实现，再次尝试编写测试：

```
@Test
public void testCalculateWordsCountOnFourWords() {
  WordCount wordCount = new WordCount();

  Map<String, Integer> actual =
    wordCount.calculateWordCounts(new File("four-words.txt"));

  Map<String, Integer> expected =
    ImmutableMap.of("hello", 3,
                    "startup", 2,
                    "people", 1,
                    "world", 1);

  assertEquals(expected, actual);
}
```

好了，第一个单元测试已经完成了，运行一下确定它会失败：

```
There was 1 failure:

1) testPrintWordsCountOnFourWords
   (com.hello.startup.wordcount.TestWordCount)

java.lang.AssertionError:
Expected :{hello=3, startup=2, people=1, world=1}
Actual   :null
```

不管是在编写实现之前还是之后编写测试，我们总是应该在编写测试之后立即检查测试是否失败，这样才能保证该测试在检查我们关注的行为时不会因为错误的原因而失败（或者通过）。此外，这样也让我们有机会检查测试失败是否会显示清晰的错误信息。

我们应该再编写几个测试用例，确保自己对问题有充分的理解。一个好的测试用例可以检查 `calculateWordCounts` 是否会忽略 to、the、and 这样的停用词。下面是一个名为 four-words-plus-stop-words.txt 的新测试文件：

```
Hello! Hello to the startup people! And hello to the startup world!
```

这个文件和 four-words.txt 基本相同，只是添加了停用词 to、the 和 and。等一下，我们怎么知道 `WordCount` 类会把 to、the 和 and 当作停用词呢？这个类最初的 API 并没有暴露出停用词，这就意味着它们是一种隐藏在内部的实现细节，这样会让测试变得更加困难，因为对这一实现细节的改变可能会让测试用例出现问题。此外，这种设计也使 `WordCount` 缺少一些灵活性，因为停用词的列表在不同的使用场景下可能是不同的。

我们的解决方法就是遵循依赖倒置原则（阅读 6.8 节了解更多信息），把停用词的列表注入 `WordCount` API 中：

```
public class WordCount {
  private final Set<String> stopWords;

  public WordCount(Set<String> stopWords) {
    this.stopWords = stopWords;
  }

  public Map<String, Integer> calculateWordCounts(File file) {
    return null;
  }
}
```

现在可以为 four-words-plus-stop-words.txt 编写一个测试用例，确保 `WordCount` 将使用我们指定的停用词：

```
@Test
public void testCalculateWordCountsIgnoresStopWords() {
  Set<String> stopWords = ImmutableSet.of("and", "the", "to");
  WordCount wordCount = new WordCount(stopWords);

  Map<String, Integer> actual = wordCount.calculateWordCounts(
    new File("four-words-plus-stop-words.txt"));

  Map<String, Integer> expected =
    ImmutableMap.of("hello", 3,
                    "startup", 2,
                    "people", 1,
                    "world", 1);

  assertEquals(expected, actual);
}
```

编写了第二个测试用例之后，另一个问题出现了：停用词是直接在测试代码中定义的，但我们处理的文本却是在文件中定义的。我们不得不在这二者之间来回切换，这样对测

试用例的理解也变得更加困难。一如既往，测试中存在的困难也引出了一个更大的设计问题：calculateWordCounts 函数没有理由去读取一个文件。对文件的读取会引入副作用（I/O），使得代码变得更难理解，也使代码缺少灵活性，因为我们可能会统计由数据库或远程 Web 服务提供的文本中的单词数量，而不使用文件所提供的文本。

calculateWordCounts 更加灵活的设计是接受一个文本 String，而不是 file 对象：

```
public Map<String, Integer> calculateWordCounts(String text) {
  return null;
}
```

这样可以使 calculateWordCounts 成为一个纯函数（阅读 6.7.3 节了解更多信息），其中一个好处是编写测试用例更容易了：

```
public class TestWordCount {
  final Set<String> stopWords = ImmutableSet.of("and", "the", "to");
  final WordCount wordCount = new WordCount(stopWords);

  @Test
  public void testCalculateWordCountsOnFourWords() {
    String text =
      "Hello! Hello startup people! Hello startup world!";
    Map<String, Integer> actual =
      wordCount.calculateWordCounts(text);

    Map<String, Integer> expected =
      ImmutableMap.of("hello", 3,
                      "startup", 2,
                      "people", 1,
                      "world", 1);

    assertEquals(expected, actual);
  }

  @Test
  public void testtestCalculateWordCountsIgnoresStopWords() {
    String text =
      "Hello! Hello to the startup people! " +
      "And hello to the startup world!";
    Map<String, Integer> actual =
      wordCount.calculateWordCounts(text);

    Map<String, Integer> expected =
      ImmutableMap.of("hello", 3,
                      "startup", 2,
                      "people", 1,
                      "world", 1);

    assertEquals(expected, actual);
  }

}
```

再次运行这个测试用例，确认可以获得清晰的错误消息：

```
There were 2 failures:

1) testCalculateWordCountsOnFourWords
   (com.hello.startup.wordcount.TestWordCount)

java.lang.AssertionError:
Expected :{hello=3, startup=2, people=1, world=1}
Actual   :null

2) testCalculateWordCountsIgnoresStopWords
   (com.hello.startup.wordcount.TestWordCount)

java.lang.AssertionError:
Expected :{hello=3, startup=2, people=1, world=1}
Actual   :null
```

应该再添加几个测试用例，比如用来测试空字符串的用例和测试各种空白和标点的用例，但是对这个例子而言，上面两个测试已经足以让我们开始接下来的工作了。现在可以开始实现 calculateWordCounts，让这些测试通过。下面是第一次尝试：

```
public Map<String, Integer> calculateWordCounts(String text) {
  String[] words = text.split("\\s+");

  Map<String, Integer> counts = new HashMap<>();
  for (String word : words) {
    if (!stopWords.contains(word)) {
      Integer count = counts.get(word);
      counts.put(word, count == null ? 1 : count + 1);
    }
  }

  Comparator<String> sortByValue =
    (key1, key2) -> counts.get(key2).compareTo(counts.get(key1));

  Map<String, Integer> sortedCounts = new TreeMap<>(sortByValue);
  sortedCounts.putAll(counts);

  return sortedCounts;
}
```

这段代码将文本根据空白字符拆分为单词，对所有单词进行循环，使用 HashMap 对不是停用词的单词进行计数，然后再使用一个带自定义的 Comparator 的 TreeMap，根据计数情况对单词进行排序。看起来这是一个很合理的实现，但我们应该运行一下测试，看看它们认为这个实现如何：

```
失败!!!
运行测试：2， 失败：2
```

看起来还有 bug，为了弄清楚哪里出了问题，来看其中一个测试失败的详细情况：

```
1) testCalculateWordCountsOnFourWords
   (com.hello.startup.wordcount.TestWordCount)

java.lang.AssertionError:
Expected  :{hello=3, startup=2, people=1, world=1}
Actual    :{Hello=2, startup=2, Hello!=1, people!=1, world!=1}
```

我们立即可以发现代码中存在一个和标点有关的 bug，因为出现了类似 Hello! 和 people! 这样的单词。看来我们不仅要根据空白字符（"\\s"）拆分文本，还要用到标点字符（"\p{Punct}"）。我们可以为这一规则表达式命名：

```
private static final String PUNCTUATION_AND_WHITESPACE =
  "[\\p{Punct}\\s]+";
```

然后在 `calculateWordCounts` 中使用：

```
String[] words = text.split(PUNCTUATION_AND_WHITESPACE);
```

运行该测试：

```
失败!!!
运行测试: 2,  失败: 2
```

看看其中一个失败的测试，弄清楚发生了什么：

```
1) testCalculateWordCountsOnFourWords
   (com.hello.startup.wordcount.TestWordCount)

java.lang.AssertionError:
Expected :{hello=3, startup=2, people=1, world=1}
Actual   :{Hello=3, startup=2, world=1}
```

可以看到标点的问题已经解决了，但现在又出现了两个新的问题。首先，代码没有正确地处理大写字母，所以“Hello”和“hello”被当成了不同的单词。其次，单词“people”消失了，但现在还不太清楚问题出在哪里。我们很容易想到用调试器或一些 `println` 语句去找出问题所在，但如果这个测试用例的错误信息不足以让我们找到出错的地方，也许表明代码在设计上可能存在一些问题。如果研究一下 `calculateWordCounts` 中的代码，它实际上执行了两个独立的任务。

- 解析单词：将一个字符串拆分成单词，并去掉停用词。
- 单词计数：计算单词出现的次数，并根据次数排序。

可以看出，该函数的职责显然太多了（阅读 6.6 节了解更多信息），导致我们虽然在测试期间发现了一个 bug，但难以判断是函数的哪一部分导致了这个 bug。因此，解决的办法并不是使用调试技巧，而是把函数不同的部分分解到独立的函数中，这样就可以单独考虑每一部分并进行独立测试。假设把 `calculateWordCounts` 分成两个函数：

- `splitTextIntoNormalizedWords`
- `countOccurrences`

第一个函数对单词进行处理，但不进行与计数有关的操作。第二个函数进行计数，但并不一定必须针对单词。如果把这些函数添加到 `WordCount` 类中，最终得到的是低内聚的 API（阅读 6.9 节了解更多信息）。有些程序员把辅助方法标记为 `protected`，从而解决了这一问题——这些方法在 API 中是不可见的，但是测试仍然可以访问它们。虽然这样总比没有好，但从内部来看，这个类仍然是松散的，所以更好的方法是把这些函数各自放到自己的类中，使之成为有聚合力的公开 API 的一部分。

例如，可以创建一个 `WordParser` 类，专门负责对文本中的单词进行处理。我们继续应用 TDD 的过程，应该先推测这个类的 API，不编写具体的实现：

```
public class WordParser {
  public List<String> splitTextIntoNormalizedWords(String text) {
    return null;
  }
}
```

我们应该对这段代码测试什么呢？可以先从四个用例入手：

- 根据任何空白或标点进行单词拆分；
- 将所有单词转换为小写；
- 处理空字符串；
- 去掉停用词。

这四项听起来很熟悉。我们只要想一下怎么去测试这个类的停用词，就会意识到需要将停用词列表注入 WordParser 类中：

```
public class WordParser {
  private final Set<String> stopWords;

  public WordParser(Set<String> stopWords) {
    this.stopWords = stopWords;
  }

  public List<String> splitTextIntoNormalizedWords(String text) {
    return null;
  }
}
```

现在编写四个测试用例：

```
public class TestWordParser {
  final Set<String> stopWords = ImmutableSet.of("and", "the", "to");
  final WordParser wordParser = new WordParser(stopWords);

  @Test
  public void testSplitTextIntoNormalizedWordsWithPunctuation() {
    String text =
      "Hello! Can you hear me? This should, um, ignore punctuation.";
    List<String> expected = ImmutableList.of(
      "hello", "can", "you", "hear", "me",
      "this", "should", "um", "ignore", "punctuation");
    List<String> actual =
      wordParser.splitTextIntoNormalizedWords(text);

    assertEquals(expected, actual);
  }

  @Test
  public void testSplitTextIntoNormalizedWordsEmptyString() {
    List<String> out = wordParser.splitTextIntoNormalizedWords("");
    assertEmpty(out);
  }

  @Test
  public void testSplitTextIntoNormalizedWordsDifferentWhitespace() {
    String text =
```

```
    "Hello\nthere!\t\tIs this     working?";
  List<String> expected =
    ImmutableList.of("hello", "there", "is", "this", "working");

  List<String> actual =
    wordParser.splitTextIntoNormalizedWords(text);

  assertEquals(expected, actual);
}

@Test
public void testSplitTextIntoNormalizedWordsRemovesStopWords() {
  String text =
    "Hello to you and all the best!";
  List<String> expected =
    ImmutableList.of("hello", "you", "all", "best");

  List<String> actual =
    wordParser.splitTextIntoNormalizedWords(text);

  assertEquals(expected, actual);
}

}
```

运行，测试全都失败了，正如我们所料：

```
失败!!!
运行测试: 4,  失败: 4
```

接下来把 `WordCount` 类原来的实现复制进去，看看这几个测试能否通过：

```
private static final String PUNCTUATION_AND_WHITESPACE =
  "[\\p{Punct}\\s]+";

public List<String> splitTextIntoNormalizedWords(String text) {
  return ImmutableList.copyOf(text.split(PUNCTUATION_AND_WHITESPACE));
}
```

坏消息是当我们用这个版本去运行测试时，四个测试仍然全是失败的。好消息是我们现在是在测试单个功能而不是整个 `WordCount`，所以这些测试的错误消息对我们会更有帮助：

```
java.lang.AssertionError:
Expected :[hello, there, is, this, working]
Actual   :[Hello, there, Is, this, working]
```

这段代码并没有处理大写字母。我们可以使用 Java 8 Streams（阅读 6.7.2 节了解更多信息）将每个单词都转换为小写字母：

```
public List<String> splitTextIntoNormalizedWords(String text) {
  return Arrays
    .stream(text.split(PUNCTUATION_AND_WHITESPACE))
    .map(String::toLowerCase)
    .collect(Collectors.toList());
}
```

再次运行测试，两个测试通过了，但仍然有两个失败：

```
1) testSplitTextIntoNormalizedWordsEmptyString
   (com.hello.startup.wordcount.TestWordParser)
2) testSplitTextIntoNormalizedWordsRemovesStopWords
   (com.hello.startup.wordcount.TestWordParser)
```

失败的测试名称昭示了问题所在——需要对空字符串和停用词进行过滤：

```
public List<String> splitTextIntoNormalizedWords(String text) {
  return Arrays
    .stream(text.split(PUNCTUATION_AND_WHITESPACE))
    .map(String::toLowerCase)
    .filter(word -> !word.isEmpty() && !stopWords.contains(word))
    .collect(Collectors.toList());
}
```

再次运行测试，所有测试都通过了。

接下来就是计算单词出现的次数。我们在运行和实现自定义的 `countOccurrences` 函数之前，应该先停下来想一想，统计数量是很常见的，也许有人已经解决了这一问题呢？实际上，在 Java 中可以轻松找到一种解决方案，那就是 Google 的 Guava 库中的 `MultiSet` 类。Guava 是一个流行的、经过充分测试且稳定的库，`MultiSet` 专用于统计，所以比简单的 `Map` 更适合用在 `WordCount` 的 API 中。

现在更新 `WordCount` 类，让它使用 `WordParser` 和 `MultiSet` 类：

```
public class WordCount {
  private final WordParser parser;

  public WordCount(WordParser parser) {
    this.parser = parser;
  }

  public Multiset<String> calculateWordCounts(String text) {
    List<String> words =
      parser.splitTextIntoNormalizedWords(text);
    ImmutableMultiset<String> counts =
      ImmutableMultiset.copyOf(words);
    return Multisets.copyHighestCountFirst(counts);
  }
}
```

这段代码将 `WordParser` 作为依赖项注入，用它来获得标准化的单词列表，然后再调用 `ImmutableMultiset.copyOf` 对单词进行计数，并调用 `Multisets.copyHighestCountfirst` 对计数结果按从高到低的顺序进行排序。整个 `calculateWordCounts` 函数现在只有三行代码。

现在更新 `WordCount` 的测试，确保一切均正常运转。

```
public class TestWordCount {
  private final Set<String> testStopWords =
    ImmutableSet.of("the", "and", "to");
  private final WordParser wordParser = new WordParser(testStopWords);
  private final WordCount wordCount = new WordCount(wordParser);

  @Test
```

```
  public void testCalculateWordCountsOnFourWords() {
    String text =
      "Hello! Hello startup people! Hello startup world!";
    Multiset<String> actualCounts =
      wordCount.calculateWordCounts(text);

    Multiset<String> expectedCounts = ImmutableMultiset
      .<String>builder()
      .addCopies("hello", 3)
      .addCopies("startup", 2)
      .addCopies("people", 1)
      .addCopies("world", 1)
      .build();

    assertEquals(expectedCounts, actualCounts);
  }

  @Test
  public void testCalculateWordCountsIgnoresStopWords() {
    String text =
      "Hello! Hello to the startup people! " +
      "And hello to the startup world!";
    Multiset<String> actualCounts =
      wordCount.calculateWordCounts(text);

    Multiset<String> expectedCounts = ImmutableMultiset
      .<String>builder()
      .addCopies("hello", 3)
      .addCopies("startup", 2)
      .addCopies("people", 1)
      .addCopies("world", 1)
      .build();

    assertEquals(expectedCounts, actualCounts);
  }
}
```

接下来运行所有的 `WordParser` 和 `WordCount` 测试：

```
JUnit version 4.11
..
Time: 0.156

OK (6 tests)
```

所有测试都通过了！遵循“测试－编码－测试”的循环，我们最终得到了整洁、可重用、经过充分测试的代码。注意，我们并不是提前规划好设计，而是让设计根据从测试中得到的迭代式反馈不断发展、逐步形成。换句话说，整洁的代码多是进化的结果，而非天才的设计。而且让代码进化也不是什么吓人的事，因为我们只要花 0.156 秒就可以运行测试，知道会不会有什么地方出问题。

当然，由于这些测试全都是单元测试，所以运行速度非常快。在现实中实现 TDD 也就意味着先要编写集成测试、验收测试和性能测试。提前考虑集成测试可以迫使自己先考虑如何部署系统以及系统之间如何相互交互；提前考虑验收测试可以迫使自己考虑是不是

正在实现正确的产品和解决真正的用户问题；提前考虑性能测试可以迫使自己考虑瓶颈在哪里，应该用哪些指标去找出瓶颈。

> 在测试过程中包含部署步骤是非常关键的，这里面有两个原因。第一，这是一种容易出错的行为，不应该手动去实现，所以当我们必须部署到真实环境的时候，脚本必须已经完全执行过。我们重复学到的一课就是，想要弄清楚一个过程，没有比尝试让它自动化实现更好的方法了。第二，包含部署步骤通常也是开发团队会无意间碰上组织中其他团队的时候。如果建立一个数据库需要花六个星期、拿到四个签名才能做到，我们现在就要知道，而不是在交付前两个星期才告诉我们。
>
> ——Steve Freeman、Nat Pryce，《测试驱动的面向对象软件开发》

提前编写测试可以提高实现彻底的测试覆盖的概率，因为这样做可以让我们增量式地编写代码。测试写起来可能很乏味，所以如果一次只需要编写一小点的话会更容易一些，即写一点测试，写一点实现代码，再写一点测试，再写一点实现代码……如果一下子编写出了上千行实现代码，再开个马拉松式的会议为所有的代码编写测试用例，效率就会很低，你会感到无聊透顶，也不会想起许多特殊的情况，因为你会忘掉之前编写的实现代码中的一些细微差别。

最为重要的是，TDD 可以迫使我们从最终结果反过头来考虑问题，帮助我们确认自己正在编写正确的代码，而不是直接跳入编码工作中，迷失在实现的细节里。我们要关注自己正在**做什么**，而不是**怎么去做**，如果我们能通过快速反馈循环的方式去做，就可以得到更高质量的设计。正所谓速度制胜。

5. 应该测试什么

测试是很重要的。它们是快速实现高质量软件最强有力的手段之一。但是测试并不是没有成本的，我们需要花时间去编写测试，在代码改变的时候要更新测试，要优化测试让它们可以快速而可靠地运行；我们还必须为后台服务器建立一个测试框架，为前端服务器建立另一个测试框架，再为客户端代码弄一个框架；也必须把所有东西都写入到 CI 任务中，让它在每次签入之后运行。大多数情况下，自动化测试是很有价值的，所以这种开销也是值得的，但在某些情况下就不一定了。

在我采访过的创业公司中，很少有在早期就彻底进行自动化测试或使用 TDD 的。这并不意味着即使这些创业公司能在测试上做得更好，也不会取得更大的成功，而是表明自动化测试并不是创业成功的严格要求。早期的创业公司可能会做 10 款产品，其中有 9 款会被抛弃（有时候 10 款产品可能都会被抛弃）。如果你编写设计良好并经过全面测试的代码，却有 90% 以上会被丢掉，价值就不是太大了。在刚刚创立的公司中，如果能够更快地从用户那里获得反馈，跳过测试有时候也可能是一种可接受的取舍。

我似乎已经听到 TDD 的狂热追随者们在尖叫着："但是没有测试的代码库是不可扩展的！"他们说的没错，但早期的创业公司必须更经常去做不能扩展的事情（阅读 3.2.4 节了解更多信息）。我采访过的几乎每一家创业公司一旦发展到了一定的规模，都要在自动化测试上进行巨大的投入才能实现可扩展——但在达到这一规模之前并不会这么做。所以关键的问题就是，我们怎么才能知道公司何时会从一个阶段进入到另一个阶段？如何判断在自动化测试上进行投入的时机？

这个答案理论上很简单：当我们不再有足够的自信可以快速改变代码的时候，就是需要自动化测试的时候了。如果我们对添加新的功能犹豫不决、在部署新的代码时战战兢兢，或者每次发布代码时出现问题的功能要比添加的功能多，那么就需要测试了。归根到底，答案就是要不断对测试策略进行评估，在几个因素之间做权衡取舍：bug 的成本、bug 的可能性以及测试的成本。

bug 的成本

如果我们在做一个极有可能会在一周内丢弃的原型，bug 的成本就是比较低的；如果我们在做一个支付处理系统，bug 的成本就非常高了——我们可不希望客户信用卡被扣两次钱或者弄错账户。在所有涉及数据存储系统（例如任何可能删除或弄坏用户数据的代码）和与安全相关的代码中（例如认证、授权、加密），bug 的成本也是很昂贵的。尽管我所采访的创业公司在测试实践上差异巨大，但每一家公司对某些代码的态度都是一致的，比如涉及支付、安全和数据的代码，这些代码从第一天起就会进行大量的测试，因为这些代码是不允许出问题的。

bug 的可能性

随着代码库规模的扩大，我们不太可能仅通过手动测试就让其保持正常运转。同样的，随着使用同一段代码的人数增加，自动化测试也成为我们避免在集成时出现 bug 的关键手段，也可以作为一种文档，记下代码应该做什么（阅读 7.2.4 节了解更多信息）。值得一提的是，一些技术问题本身就很复杂，所以需要更多的测试才能够正确地解决这些问题。例如，如果我们在解决复杂的数学问题或编写自己的分布式一致性算法，可能就需要编写自动化测试才能很好地解决问题。

测试的成本

现在建立单元测试几乎是免费的，几乎所有编程语言都有高质量的单元测试框架，几乎所有构建系统也都内置提供了对单元测试的支持。单元测试的成本如此之低，测试运行如此之快，测试对代码质量和代码正确性的改善和提升如此之高，所以几乎任何时候都应该编写单元测试。换句话说，集成测试、验收测试和性能测试的建立成本是逐步提升的。这一类测试多与具体的使用场景有关，取决于我们如何自动化部署代码、如何初始化数据存储并把一切连接起来。不管怎样，这些都是我们需要去解决的有价值的任务，但在某些情况下，其开销也许超过了收益[2]。

6. 自动化测试的最佳实践

尽管 TDD 是黄金标准，但几乎所有类型的自动化测试都比手动测试更加出色。如果我们所处理的东西并不符合 TDD 模型[3]，在编写代码之后立刻编写测试仍然也是有价值的。如果我们正在使用没有任何测试的现有代码库，这就是添加测试的最佳时机。如果我们正在修复一个 bug，上手的最好方式就是编写一个能够产生这一 bug 的失败测试（测试驱动 bug 修复）。这样可以证明我们理解了这个 bug 的成因，把它修复之后，剩下的就是一个可以防止 bug 又回来的自动化测试。

注 2：对于客户端代码的测试来说更是如此，比如 Web 应用和移动应用。这些测试很容易出现运行缓慢和维护困难的情况，因为最小的 UI 改变也可能引起测试失败。另外，还有少数的自动化工具可以对某些类型的任务进行手动测试，比如验证 UI“看起来正不正常”。

注 3：TDD 不适用的最常见场景就是“试探性编码”，即我们并不知道所做的是什么，仅仅通过编码和数据的尝试去试探问题的空间。如果我们不知道期望的结果是什么，也就无法为它编写测试。

另外，自动化测试还应该能够很好地实现自动化。也就是说，我们应该能够用一条命令去运行所有的测试，不需要手动介入，最好是能够让它成为构建过程的一部分（阅读 8.3.3 节了解更多信息）。单元测试的运行时间不应该超过几秒钟，而所有的集成测试、验收测试和性能测试则不应该超过几分钟[4]。需要花好几个小时运行的庞大测试包几乎是没有价值的，因为它一天只能给我们一次反馈，可能只比 20 世纪 60 年代彻夜在大型机上提交穿孔卡片的程序员高效一小点。我们要么让测试能够快速运行（降低依赖，让测试并行执行），要么就把它们删除掉，因为运行缓慢的测试包通常来说弊大于利（阅读 8.2 节了解更多信息）。

唯一比运行缓慢的测试包还要糟糕的是不可靠的测试包。间歇性出现失败的测试或不确定的行为是有害的，因为它们会降低我们修改代码的信心。如果我们不能马上修复短暂出现的失败，就会对整个测试包的价值造成损害。如果总是看着测试失败时不时出现又消失，你就会习惯性地忽略失败，就像喊着狼来了的男孩一样，没人知道你什么时候会遇到真正的问题。所以，请立即修复或删除不可靠的测试。事实上，我们从一开始就要提防引起大多数短暂测试失败的原因——时间，从而避免编写这样的测试。在编写依赖于 UI 的异步事件、随机数、系统时钟、最终一致性的数据存储、具有生存时间的缓存或者多线程程序的执行顺序的测试时，我们要特别注意。

7.2.2 代码分离

> 软件开发并不是在图纸上、IDE 中或设计工具里进行的，它是在你的脑海中进行的。
>
> ——Venkat Subramaniam、Andy Hunt，
> 《高效程序员的 45 个习惯：敏捷开发修炼之道》

本书的前面讨论了产品的设计必须简单，才能适应人类大脑的局限性。如果产品中包含了太多的信息——太多的功能、太多的文本、太多的按钮、太多的设置——它就会让人的记忆不堪重负，产品也会变得无法使用（阅读 3.1.2 节了解更多信息）。同样的原理也适用于代码。如果我们在一个地方放了太多代码，就会让程序员的记忆不堪重负，代码库也会变得更加难以理解。其实也不需要太多代码，只要几千行，大概一个小型 iPhone 应用或 JavaScript 库那样的规模，大多数人就已经无法记住了。

尽管大多数程序员认为他们的任务就是编写代码，但在现实中，代码却是许多时候的敌人。你拥有的代码越多，前进的速度就越慢。缺陷密度（每千行代码的 bug 数量）会随着项目规模的增加而显著增长，所以如果项目的代码行数是两倍，bug 的数量通常就不止两倍，而规划、实现、测试和维护的困难程度也不止两倍。想象一下，如果你把编写的每一行代码都打印在纸上，必须装在背包中随身携带，随着代码库的增长，成堆的纸张将渐渐拖慢你的步伐，直到你裹足不前。

对于代码库来说，最糟糕的事情就是规模的扩大，所以我们应该努力让代码的规模尽可能小。一方面可以把尽可能多的工作委派给基础代码之外的开源和商业函数库来实现（阅读 5.3 节了解更多信息），另一方面可以使用编码风格简洁、减少样板文件引用和重复

注 4：持久性测试是个例外，因为它们是专用于了解系统在长时间运行下对负载的应对能力的，应该是在构建过程之外运行的。

的编程语言来实现（阅读第六章了解更多信息）。但在某种程度上，随着公司和产品规模的扩大，基础代码也不可避免会随之增长，我们还需要采用新的技术去应对。

最好的解决办法就是将代码分离成多个“片段”。通过这种方式，我们可以一次只考虑一个片段，并可以放心忽略其他片段。这就是所谓的**抽象**。其实我们一直在做着这样的事，哪怕是在编程之外。例如，当我们凭着记忆去画东西时，比如画一个橘子，你会发现自己脑海中的图像是一个简化版的橘子，它已经丢失了许多的细节（阅读 3.1.1 节了解更多信息）。在大多数场合，对橘子的这种抽象已经足够了，因为我们不需要察觉一个橘子的每一个微小细节，比如为了找到橘子吃，我们不需要察觉它精确的颜色和纹理。一个出色的抽象应该满足两个属性：信息隐藏和可组合性。

信息隐藏意味着抽象应该比其背后的细节更加简单，就像脑海中的图像要比真正的水果更加简单。我们可以这样想，抽象的表面积（即它暴露给外部世界的接口）应该比其体积（即它在内部隐藏的实现细节）更小。在现实当中，物体的表面积会随着其大小的平方而增长，而体积则是随着其立方而增长。同样，在软件领域，实现细节中信息量的增长是非常快的，为了防止大脑不堪重负，我们需要能够表示更小、更简单的接口的抽象。

可组合性意味着可以把多个抽象组合起来，从而得到新的抽象，这一抽象反过来也仍然能够和其他抽象组合。这样的话，我们就可以从简单的片段开始，增量式地构建出复杂的抽象，一次一个抽象层——有点儿像用乐高积木搭房子，可以从数百个独立的小积木块开始，每一块都很简单。但是我们是记不住那么多积木的。因此，要用十多块积木组成地板，另一些组成墙，其他十几块组成天花板。这样的话，就不需要想着要怎么去用几十块积木，只需要记住少数几种即可：地板、几面墙和天花板（实现了信息隐藏）。这样做的好处还不止于此——我们把积木搭成房子的第一层后，反过来这一层又可以和其他的乐高构件组合起来（例如楼梯、其他的楼层、车库），从而搭建出越来越大的构件（可组合性）。

编程中有许多种类型的抽象，这里只介绍以下最常见的三种：

- 接口与模块；
- 版本化构件；
- 服务。

1. 接口与模块

大多数编程语言都允许定义**接口**，用于指定一段代码能够执行一组操作；也允许定义**模块**，将有关联的结构组成一组。根据语言的不同，接口也许是一个函数签名、类中的公开方法、抽象的接口或特质（trait），而模块也许是一个包、命名空间或者一个库。正如第 6 章介绍的，精心定义的接口和模块是实现整洁代码的基石。

2. 版本化构件

模块与接口都是分离代码的有效方式，但是随着代码的不断增长，各种模块很容易“缠绕”在一起，代码会开始变得像满是线的盒子。你伸进去拉出一根线，但所有东西都相互缠绕，最终几乎会把整个盒子里的所有东西都拉出来。虽然有一些编程实践可以用来降低这种耦合（阅读 6.8 节了解更多信息），但随着代码越来越多，这样的结果几乎是不可避免的。

一种可能的解决方法就是分解代码，不是让模块依赖于其他模块的**源代码**（**源代码依赖**），而是让它们依赖于其他模块发布的**版本化构件**（**版本化依赖**）。就像大家目前使用开源库的方式，想要在 JavaScript 代码中使用 jQuery 或在 java 代码中使用 Google Guava，我们不去依赖这些开源库的源代码，但是要依赖于它们所提供的版本化构件，比如 jquery-1.11-min.js 或者 guava-14.0.jar。我们也可以在自己的代码中使用同样的方法，通常来说，这意味着要对构建系统进行修改，每次构建之后要发布版本化的构件，实现版本依赖而不是源代码依赖（阅读 8.3.2 节了解更多信息）。事实上，只要把所有的源代码依赖迁移到版本依赖，我们甚至可以将每个模块的源代码放到一个单独的代码库中，实现隔离开发（阅读 8.3.1 节了解更多信息）。

过去，LinkedIn 经常把所有代码都放入单独的一整个代码库中，所有模块之间都存在源代码依赖。但在它发展到数千个模块、数百万行代码之后，就暴露出了一系列问题（阅读 8.2 节了解更多信息）。所以在过去几年里，这家公司一直在将它们的代码分解到多个代码库中，相互间存在的是版本依赖。Twitter 走的则是另一条路：它一开始使用许多独立的代码库，相互间也是版本依赖，但现在它正尝试把所有东西合并到一个存在源代码依赖的单独的代码库中。为什么这两家公司朝着两个相反的方向前进呢？

答案就是其中存在若干非同寻常的权衡因素需要考虑。下面列出了使用存在版本依赖的多个代码库的一些优点。

隔离
: 有了版本依赖，在你决定升级到新的版本之前，我们可以将变化与其他模块隔离开，也不存在对不相关的模块进行修改导致构建出现问题或者引起代码出现 bug 的可能性。

耦合
: 把所有的代码都放在独立的代码库中，只允许显式的、版本上的依赖，这样更容易降低耦合。此外，也可以更容易地让代码开源，因为独立的、版本化的代码库是所有开源项目都在使用的模型。

构建时间
: 如果所有代码都放在一个代码库中，那么添加的每一个模块都会（至少是）线性地增加构建时间。一旦代码量达到数百万行，一次构建可能会花好几个小时。我们可以把每一个模块放在一个独立的代码库中来缓解这一问题，这样每个模块的构建时间可以保持相对恒定和快速。

以下则是使用存在版本依赖的多个代码库的缺点。

持续集成
: 对于版本依赖，我们无法解决依赖项中的 bug 和不兼容性问题，除非升级到新的版本，但这也许是在问题出现后的几个月。换句话说，我们会失去持续集成的许多优点（阅读 8.3.3 节了解更多信息）。

依赖地狱
: 对于版本依赖，可能会存在许多不同类型的依赖性问题：依赖项冲突、循环依赖、钻石依赖，等等。向后兼容已成为所有模块 API 的严格需求，升级到新版的库可能会很痛苦，也会耽误时间。

全局修改

对于多个代码库而言，进行全局性的修改是很困难的，这样的任务也不可能原子性地完成。我们必须使用特定的工具在所有代码库间搜索，签出匹配的代码，更新每个代码库中存在依赖的代码及其版本，再尝试将修改提交回去。我们在应对依赖地狱时一直要进行这样的操作。

一般情况下，如果代码是由若干独立的模块组成，并且大部分修改都在单独的模块**内部**（你的公司类似几个独立的开源项目），那么具有版本依赖的多个代码库可以让你更快地前进。然而，如果你需要定期对许多模块进行全局性地修改，存在源代码依赖的单个代码库将会是更好的选择，特别当你在工具上已经投入了很多，消除了构建速度缓慢这样的不利影响时（例如 Google 使用了一个庞大的分布式集群编译和测试代码）。

3. 服务

另一种分离代码的方法就是把相关的功能组成**服务**，即把某类功能放到一个进程后面，通过消息去和进程通信而非进行函数调用。通过独立的服务构建技术栈有许多不同的模型，包括面向服务架构（Service Oriented Architecture，SOA）、微服务和 Actor 系统。例如，在 LinkedIn 中，我们的架构类似于微服务，一些分开的服务分别用于存储个人资料数据（即每名用户的姓名、教育和经历的详细信息）、公司数据（即姓名、位置和公司图片）、云数据（即用户与公司的关联信息），等等。每一个服务都由不同的团队拥有，在分开的代码库中独立开发，部署在分开的硬件上，通过 RESTful HTTP 调用处理与其他服务的通信。

就像对代码进行分离存在许多权衡取舍，把代码分离成多个服务也同样面临许多权衡取舍。以下是服务的一些优点。

隔离

能够构建、测试和部署一个小型、独立的服务而不需要担心代码库的其余部分，这是很让人高兴的事。另外，服务的边界也能够很好地起到代码所有权边界的作用，让团队能够彼此独立，这对于发展中的公司的工作划分是很重要的。

技术不可知论

模块只可能在一种语言内共享（例如 Java 模块只可能从其他 Java 代码调用），但我们可以使用任何想要的语言去实现服务，因为远程端（例如 HTTP）可以被任意语言使用。如果你的公司需要使用多种编程语言（例如收购或者要迁移到新技术栈的时候），服务也许就是实现功能共享唯一的选择。

性能

如果产品有足够的流量或者数据，多个服务也许是让性能变得可接受的唯一方式。举例来说，一些功能也许需要一台有很多内存的服务器，而其他一些类型的功能则需要有较快的 CPU 的服务器集群。如果将这些功能划分到不同的服务上，我们就可以对每个功能进行独立扩展。

以下则是服务的一些缺点。

操作的复杂性

我们现在面对的是不同类型的服务，每个服务都有自己的需求和技术，不是只部署一种类型的服务就行了。此外，这些服务相互间会以复杂的方式对话，所以我们需要一

些机制去实现服务发现、负载均衡、分析调用曲线——所有这一切在实现和维护上都是十分昂贵的。

错误处理

虽然调用本地函数都是成功的，但在调用远程服务时却可能会失败，这也许是网络问题引起的，也许是服务停止了，也许仅仅因为花的时间太长了。使用服务之后，所有代码都需要对新的错误类型进行处理。

性能开销

远程调用所花的时间要比本地函数调用多几个数量级，所以我们必须重新组织代码，最大程度减少潜在的开销（例如批处理、去除复制、预获取、缓存）。在对远程请求进行序列化和反序列化的时候，也存在 CPU 和内存方面的开销。

I/O

如果使用阻塞 I/O 进行远程调用，就必须对每个服务的线程池进行管理，这么做会增加操作开销，并可能引起性能上的问题。如果使用的是非阻塞 I/O，可以在一定程度上避免这样的问题，但又必须使用基于回调或承诺实现不同的编码风格。

向后兼容性

我们无法删除或重命名一个服务的 API，哪怕是 API 中一个单独的参数，因为现有的客户端很可能仍然在使用老版本。这意味着我们无法简单地对一个服务的 API 进行重构，而是必须进行版本的更替，并可能要在很长一段时间内去维护老版本，从而增加了维护工作的开销。

一般来说，服务就蕴含着大量的开销，所以除非没有其他选择，最好还是避免使用服务。也就是说，如果真的需要让团队能够彼此独立地工作、使用不同的编程模型，或者单独一个统一的应用已经无法再满足负载（阅读 7.3 节了解更多信息），就应该转而使用服务——但也只有在能投入相当大的精力去维持其运转时才能这样选择。

7.2.3 代码评审

> 这本书的每一页都已经被编辑检查过了。为什么呢？因为即便你是最聪明、最有才华、最有经验的作者，你也不能校对自己的作品。你对这些概念太熟悉，这些单词已经在你脑海中转了那么长时间，你已经无法设身处地地为第一次听到这些词的人着想。编写代码也一样。事实上，如果没有独立审查，就不可能写出美妙的文章，我们当然也就不可能孤立地写出代码；代码的每一个微小细节都必须是正确的，这样的代码便也是人类的一种文章！
>
> ——Jason Cohen，WP Engine、Smart Bear Software 创始人

如果整洁的代码就是让其他程序员可以更容易地理解你的代码，那么实现这一点的最好做法就是把代码展示给其他程序员看。让代码被其他人评审是捕捉 bug 最有效率的手段之一，一项对有着 11 名程序员的团队的研究表明，在他们开发出来的没有代码评审的程序中，平均每 100 行代码就有 4.5 个错误，而那些有代码评审的则平均每 100 行只有 0.82 个错误，错误率下降 80% 以上。另一项研究表明，虽然单元测试和集成测试均可以捕捉到大概 30%~35% 的 bug，但设计评审和代码评审可以捕捉到 55%~60% 的 bug。

除了捕捉 bug 之外，代码评审还有另一个重要的好处：它们是在整个团队间传播知识、文化、培训以及所有权感的一种高效机制。所有参与代码评审的人都会享受到这种好处：高级工程师可以利用代码评审去指导初级工程师，初级工程师可以通过参与代码评审去学习代码并贡献一些重要的问题。如果新的开发人员无法理解这些代码，也许是因为开发人员经验不足，但也有可能因为代码是混乱的。

代码评审分为四种类型：设计评审、结对编程、提交前评审和静态分析。

1. 设计评审

在开始一个大型新项目之前，先做出设计并从团队中收集反馈是比较好的做法。这是一种让团队有机会提出改进建议、潜在问题，讨论如何让你的工作步调和他人一致的好方法。我们不需要花 3 个小时去编写一份 300 页的规格说明书，但是花几个小时考虑清楚思路可以节省后面几个星期的编码时间（阅读 7.2.4 节了解更多信息）。我们要保持设计评审轻量化、对他人友好，确保所有人都参与进来：高级工程师应该让他们的设计接受初级工程师的评审，反之亦然。

> 我们在 Coursera 做设计时，仅用 Google Docs 去记录所有东西。你可以大概写个框架，加上添加请求、添加评论的建议，描述你所想的东西，还有一堆开放式的问题。然后把它丢到整个技术团队的墙上，他们会在上面吐槽，加上各种评论。这样你就可以从同事那里得到大量的反馈，每个人也都能够提供他们的输入。这会让你更加出色，做出更好的设计。我发现让初级工程师去观察（甚至参与）设计过程并做出更好的架构设计是非常有价值的。最终，所有的评论都会得到解决，你也得到了一份大概的设计草图。整个过程是异步的，一般不超过两天。
>
> ——Nick Dellamaggiore，Linkedin、Coursera 软件工程师

2. 结对编程

结对编程是一种开发技术，让两名程序员在一台计算机前面工作。一人是驾驶员，负责编写代码；另一人是观察员，负责评审代码，并在更高层次上思考程序。这两名程序员定期交换角色。

这样可以形成一个持续不断的代码评审过程。习惯这种方式需要花些时间，但有一位开发人员在旁边的话，你就会一直关注如何让别人清楚地知道代码在干什么，也拥有第二双眼睛去捕捉 bug。尽管结对的开发人员在程序上平均要比单个开发人员多花 15% 的时间，但产生的代码中的瑕疵大概要少 15% 左右，设计质量也更高。此外，由于对代码熟悉的人已不止一个，未来对代码的维护也变得更容易[5]。

我们不需要对每一行代码都实行结对编程，但每当开发一些棘手或关键业务的代码时都应用结对编程是一种很有价值的做法。结对编程也是面试应聘者和检验新入职员工的一种好方法。

3. 提交前评审

一旦完成了一段代码的编写和测试，我们就要准备好提交代码，先把代码提交给团队进

注 5：结对编程可以提高你的巴士系数（bus factor）。巴士系数是衡量你失去多少人后就无法继续一个项目的指标——由于这些人被巴士撞了，或者由于某些不那么极端的原因，比如去度假或者离开公司。巴士系数越高越好。

行最终评审是一种好的做法。提交前评审可以让每个人都有机会看到我们所构建的东西、提出问题并找到错误。我们应该使用在线工具去跟踪代码的评审注解，这样才能在后续调试或尝试理解一段代码背后的上下文时找到这些注解。出色的代码评审工具包括 GitHub（把代码评审作为 pull 请求过程的一部分）、ReviewBoard 和 Phabricator。

4. 静态分析

除了让人们评审我们的代码，使用自动化的工具去检查代码也是个好主意。对于编译型语言来说，最重要的工具就是编译器本身，但每一种编程语言都有一些静态分析工具和 linter 工具。这些工具可以帮助我们识别出一些常见的源代码 bug 的来源，发现代码风格的问题，检测重复的代码和没有使用的代码，对一些复杂的指标进行计算以找出需要重构的代码，找出使用了不安全特性（例如 `eval`、`goto`）的代码，识别出潜在的安全漏洞，发现潜在的内存泄漏。

大多数静态分析工具可以通过命令行运行（例如在签入之前运行）并集成到构建系统中。严重的静态分析错误会引发构建失败。理想情况下，我们也可以在编码的时候运行静态分析工具。许多 IDE 和文本编辑器都内置了静态分析功能，所以可以立即得到输入的反馈。例如，JetBrains 展示了使用带静态分析功能的、合适的 IDE 可以引发 Apple 公司恶名昭彰的 gotofail 的 bug（阅读 6.2 节了解更多信息），如图 7-2 所示。



图 7-2：IDE 中的静态分析功能

5. 代码评审最佳实践

代码评审是每一个公司都应该应用的必备实践，但要更好地实施也要遵循若干指南。

首先，为代码的每个部分指定一个所有人是一个好做法。所有人并不是唯一允许修改这部分代码的人，但他们有责任知道这段代码如何工作，并且要保持它运行。这意味着他们要决定可以对代码做什么样的修改、谁可以进行评审、必须强制执行什么编码标准、何时可以部署代码。

其次，需要提前将代码评审指南写下来，营造良好的代码评审文化。这些指南应该包含评审者期望掌握的事项的备忘录，比如代码是否容易阅读、是否遵循团队的编码约定、是否包含测试，这些指南也应该定义实施代码评审的代码。代码评审并不是炫耀自己的知识，取笑他人的代码或者责怪他人。它是一种学习的手段，能让团队中的所有人变得更加出色，让所有人都感觉（及表现得）像主人一样。为了让所有人都达成共识，代码评审指南应该定义什么代码必须被评审（例如是所有的代码还是实现关键任务的那部分），什么时候代码必须被评审（例如是每次提交前、合并分支前还是在每周的评审会议上），谁的代码必须被评审（例如所有人，不管级别有多高），谁负责给出评审意见（例如所有人，不管级别多低），什么类型的意见是合适的（例如花时间标注代码好的一面，而不仅仅是不好的一面；永远不要侮辱提交者本人；不要害怕承认你不理解某些内容）。

第三，保持小规模的评审。10 行代码的修改是很容易评审的，但是 1000 行代码的评审几乎是不可能的。这就意味着我们应该鼓励开发人员进行小修改，增量式地提交，而这恰好也是减少 bug、合并冲突和后期集成问题出现概率的好方法。

7.2.4 文档

文档对于代码规模的扩展和开发团队都是必不可少的，即便团队只有一个人也是如此。编写文档就像编写自动化测试，可以显著提升代码的质量。如果你强迫自己从用户的角度去看待项目，将会得到更好的设计。如果你花一个小时用文字去描述正确的解决方案，你就能节省下用代码编写错误解决方案所需的一个星期。

这里说的“文档”不仅指参考手册，还指所有帮助学会软件的东西，包括书面文档（Readme、教程）、代码文档（类型系统、注释）和社区文档（Q&A 网站、邮件列表）。每种类型的文档都在解决不同的问题，所以大多数项目都应该混合包含各种类型的文档。

1. 书面文档

书面文档由 Readme 文件、教程、参考手册和项目网站构成。

Readme 文件是代码中最重要的文档。它概括项目的任务、描述项目的作用、展示例子并解释如何开始使用、如何为项目做贡献，以及从哪里获取更多的信息。后文将介绍有关 Readme 文件更详细的信息。

如果 Readme 文件是把用户领进门，**教程**就是告诉他们如何四处走走。教程的目的是带着用户，一步一步经历各种典型的开发流，强调项目的一些惯用开发模式、最佳实践和特性。教程不需要太过深入地向用户介绍，每个步骤只要向用户提供可以找到更多信息的链接即可。我们也可以把小型、简单项目的教程塞到 Readme 文件中，但是较大的项目

则需要用到 wiki、博客、单独的网页、幻灯片甚至视频。如果还想更进一步，可以尝试做一份交互式的教程。例如 Go 语言之旅教程可以让人们直接在浏览器中使用 Go 编程语言，什么都不需要安装。对大多数开发人员来说，实践都是最佳的学习方式，所以能够让开发人员一步一步参入其中的指南将是一种强大的学习工具。

在新用户通过 Readme 文件步入大门并跟着教程走一段路之后，他们将了解到足够多的信息，会开始提出一些问题，而这就是**参考手册**所要起的作用。这种文档可以深入涵盖所有主题。记住，参考手册的作用就是回答问题，所以要确保以一种容易搜索和导航的方式去组织信息。

最后，**项目网站**也可能是进行市场推广时的一种好的文档形式。我们可以给自己的项目安个家，让它具有定制的外观和感觉，并提供可链接、可分享、可搜索的内容。为项目搭建网站最简单的方式就是使用 GitHub Pages：在 GitHub 上创建一个 repo，把一些静态 HTML 放入其中，提交代码，让首页指向 github.io 的域就可以了。

2. 代码文档

实际上根本没有什么"自我文档描述的代码"，代码只能表达你所做的，无法表达它本来应该做什么和为什么要这么做，所以每个项目都需要书面文档。也就是说，代码本身是关键的信息来源，因为它展示了实践工作的方式。代码文档最重要的因素则是注释、类型系统和示例代码。

注释是存在于代码中的书面文档，整洁的代码不需要很多注释，我们需要表达的几乎所有东西都应该由代码本身去传达。如果遇到一些无法理解的代码，解决的办法并不是添加解释性的注释，而是对代码进行重构，直到它能够以自己的术语被理解。典型的例子就是有人在一个很长的方法的主体中，每 10~15 行就插入一条注释，帮助阅读的人弄清状况。但更好的解决方案是将代码分解成多个方法，每个方法都有清晰的签名（方法名、参数名和类型），这样大部分注释就没有必要了。可以这么说，注释的需求永远不会完全消失，因为根据编程语言的不同，某些类型的信息是不能够由代码来表达的，比如背景信息（例如"这个函数基于某篇论文使用了算法 X……"）、对输入的假设（例如"初始参数必须是非负整数"这样的前提条件）、所提供输出的保证（例如"返回值必须永不为空"这样的后置条件）、对任何副作用的解释（例如"该函数会将上传数据存放在临时文件中"）以及对一些无法改进的难看或不直观代码的解释（例如"这是应对 bug XXX 的变通做法，直到下一个发布版本才能够修复"）。

> 注释的正确用法是弥补代码本身表达上的失败。注意我用的是"失败"一词，这也就意味着，使用注释总是面临着失败。但我们又必须使用注释，因为不可能一直在没有注释的情况下解决自我表达的问题，但是使用它们并不值得称赞。
>
> ——Robert C. Martin，《代码整洁之道》

可以减少代码对注释需求的一个因素就是**类型系统**。在静态类型语言中，类型系统不仅能自动阻止了某一类型错误的发生，也降低了需要编写的文档数量。举个例子，看看 Java（这是一种静态类型语言）中以下函数的类型签名：

```
public String convertCsvToJson(String csv)
```

下面是 Haskell 中同样函数的签名，这也是一种静态类型语言，比 Java 有着更强大和更

严格的类型系统：

```
convertCsvToJson :: String -> String
```

以下则是 JavaScript 中同样函数的签名，这是一种动态类型语言：

```
function convertCsvToJson(csv)
```

对于输入参数 `csv`，该函数期待得到的是什么类型的值？在 Haskell 和 Java 中，我们可以清楚地看到函数期望得到的是 `String`。在 JavaScript 中，我们并不知道函数期望的是什么类型的值：它可能想要 `file`、想要 `String`、想要 `function`，甚至还可能根据参数的类型或数量有不同的表现；除非函数的作者在文档中说明，否则是不会知道的。那么，这个函数又会返回什么类型的数据呢？在 Java 中，它返回的是 `String`，也可能是 `null`；在 Haskell 中，我们知道它返回的是 `String`（Haskell 并没有 `null`）；但在 JavaScript 中，我们无法知道函数返回什么：它可能是 `String`、可能是 `file`，或者就是一个 `void` 函数，根本不会返回任何值。除非函数的作者在文档中说明，否则我们也无从知晓。最后的问题是，这个函数有没有任何副作用呢？在 Haskell 中，如果你执行任何 I/O，编译器都需要你返回名为 `IO` 的类型（阅读 6.7 节了解更多信息）。单从函数的签名，我们看到的是它返回 `String` 而不是 `IO`，所以可以自信地认为它没有副作用[6]；另一方面，对于 Java 和 JavaScript，我们无从知道该函数是否会在内部修改全局变量、写入磁盘或者发射导弹，除非作者在文档中写明。

就文档而言，静态类型系统明显是胜者。因为开发人员可能太懒，不会为每个方法都写上注释，而且他们所写的注释也很容易过期，而我们从类型系统中获得的信息是受编译器强制实行的，所以永远都是可用的，也一直都是正确的。随着类型系统变得更加强大，甚至会在更大程度上减少注释的需要。举个例子，支持**依赖类型**的语言，比如 Idris，允许我们定义“比 0 大的整数”或者“有两个元素的列表”这样的类型，编译器可以在构建期间强制执行这些约束（不需要开发人员在注释中把它们列为前提条件）。

这一切都表明，不管类型系统多么出色或者我们写了多少文档，都无法强迫开发人员去阅读手册。有些开发人员更喜欢通过例子来学习——这不过是表达他们喜欢复制和粘贴的礼貌说法。因此，每个项目都应该包含清晰、惯用的示例代码，而自动化测试也是一种特殊的示例代码。测试可能和文档一样有用，因为它们展示了代码在大量使用场景和特殊情况下的预期表现。虽然书面的文档可能会过期，但只要测试能够通过，就可以确保它是准确的。

3. 社区文档

如果项目有相关的社区，那么该社区就是另一种丰富的文档来源，其形式包括项目管理工具、邮件列表、Q&A 面板、博客文章和讨论。举个例子，大多数团队都在使用 bug 跟踪软件（例如 JIRA、bugzilla、GitHub issues）以及一些项目管理软件（例如 Basecamp、Asana、Trello），这些系统都包含了许多与项目相关的有价值的信息：以前你做了什么、现在你在做什么、以后你要做什么、bug 发现、bug 修复，等等。我们在搜索项目信息时，无意间发现一份 bug 报告或者一个老的 wiki 网页并不罕见，特别是开源项目的话，

注 6：在 Haskell 中，有一些方法可以执行 I/O 或产生副作用而不改变的函数签名，比如使用 `unsafePerformIO`。但是正如其名称所隐含的，这是一种不安全的操作，不符合习惯用法，也很少使用。

到处都是这些公开的信息。

来自 Stack Overflow 这样的 Q&A 网站的讨论和 Google Groups 这样的邮件列表也非常频繁地出现在搜索结果中。对于内部和专有的项目，我们也可以使用内部的邮件列表，维护一份 FAQ 或者部署一个内部的像 Stack Overflow 这样的 Q&A 网站。但即便是最出色的文档也不能够回答所有的问题，所以培养社区网站可能是让软件变得可学习的关键一步。经过一段时间之后，这样的网站可能会成为项目文档最重要的部分，因为它们天生就是处理那些难住开发人员问题的地方。

最后，对于流行的开源项目而言，最佳的文档其实是博客文章和终端用户的讨论，因为他们会揭露出什么是真正可行的，什么又是不可行的。它们对项目来说也是一种很好的推广，因为这清晰地表明了有其他人正在使用这些项目。如果你的项目是开源的，培育一个相关的社区可能会给你带来巨大的回报。通过良好的文档、定制的项目页面、讨论和碰头会等形式，在项目的“市场推广”上做些小投入，可能给我们带来免费人力、更整洁的代码和更好的品牌这样巨大的收益（阅读 12.2.3 节了解更多信息）。

4. Readme驱动开发（RDD）

在前文中，我们介绍了测试驱动开发（TDD），即在实现代码之前先编写测试。对于一个新项目来说，甚至在编写测试之前就有一个应该要做的步骤：编写 Readme 文件，这就是所谓的 Readme 驱动开发（readme-driven development，RDD）。不要把它和瀑布过程的“提前设计一切”混为一谈，后者要花好几个星期做出一份 300 页的规格说明书，描述清楚每一个细节。对于 RDD 而言，只要花一个小时想清楚整个项目，最重要的是，在开始编码之前写下自己的想法。

RDD 使我们在迷失于实现的细节之前，先迫使自己思考要实现的东西，从而帮助自己确认所做的是正确的东西。你也许认为自己知道要做什么，但是在你把脑海中萦绕的模糊的想法写到纸面（或数字媒介）上的过程中，会发生一些神奇的事情。写是一种更加严密的思考形式，肯定可以暴露出计划中的瑕疵。在才写出几段文本的时候就去修复这些瑕疵，比编写了几千行代码之后再去修复要容易得多。

如果提前写出了 Readme 文件，就能够随着项目的实现递增式地填充各种细节，使得文档编写过程不那么痛苦，因为我们可以一次只写一小部分，而每一部分的信息在我们的脑海中都是新鲜的，而非在项目的最后再去进行一场文档马拉松，苦苦挣扎着记下所有细节。此外，当我们和他人一起共事时，Readme 文件本身也是无价的工具。如果让 Readme 文件在团队中流转，就有了一些可以讨论得很具体的东西，特别是可以用来进行设计评审（阅读 7.2.3 节了解更多信息）。而且只要所有人都达成了共识，Readme 文件也可以作为一份谁应该干什么的文档说明。

为了让大家更清楚为什么 RDD 如此有价值，我们来完整地看看 Readme 文件的各个部分，包括推销说辞、示例、快速入门指南和项目组织细节。

推销说辞

在 Readme 的顶部，要非常简洁地解释项目是做什么的，以及为什么应该使用它。第一点可以促使你清楚自己正在实现**什么**；第二点促使你证明**为什么**要实现它，而不是使用代码中已有的库、使用开源的库，或者彻底实现别的东西（阅读 5.3 节了解更多信息）。如果我们无法证明别人为什么应该使用它，我们可能也不应该实现它。

示例

在推销说辞之后，我们要展示几个代码片段、UI 原型、截图或者架构图，演示如何使用该项目，也让人们有机会提前了解用户体验。不幸的是，许多程序员都会跳过这一步骤，直接开始进入实现。他们会和最棘手的代码做斗争，把代码一层层堆积起来，一点一点，正如杂草爬上丛林中的大树，它们将蜿蜒而上，直到某一刻冲破树冠，第一次把代码暴露到阳光之下——暴露给用户。无论杂草的边缘多么混乱，都会变成用户体验。大多数情况下，这种 API 或者 UI 是完全不可用的，但现在去修复已经太迟了，因为有太多的树枝、树根和分叉以及代码，无法进行有意义的改变。这就是为什么应该先从用户界面开始（阅读第 3 章了解更多信息），通常来说这既是项目最难的部分，也是项目是否成功最决定性的因素。

快速入门指南

但只考虑用户界面还不够。在列出了几个例子之后，接下来就是快速入门指南，用来解释如何部署项目以及如何开始使用项目。这也会促使我们考虑项目应该如何封装、它具有什么依赖项、需要什么类型的配置、在开发和生产环境中是如何工作的，不要到最后再来做这些决定，这些都是使用产品的体验中不可或缺的，要正确实现它们总是会比我们所预期的时间更长。

我最喜欢拿 Spring Framework 上实现简单 Web 应用的快速入门指南和 Node.js 的做比较。Spring 的教程要花 15 分钟（在没有犯任何错误的情况下）并需要十多个步骤，在这个过程中要创建 8 个文件夹和文件，用 2 种编程语言编写 88 行代码（是的，我数过）。而 Node.js 只需要花 2 个步骤和 15 秒：从 Node.js 主页复制 6 行代码并粘贴到电脑上，然后运行。现在你还会对 Node.js 成为有史以来发展最快的开源项目感到惊奇吗？快速入门指南就是你的第一印象，在这一点上务必要做对。

项目组织细节

这一部分要解释清楚项目的运作方式。代码放在哪里？代码是如何组织的？任务和 bug 是如何跟踪的？他人如何为项目做贡献？法律方面是如何考虑的（许可、版权）？如果你想和他人一起运作这个项目，提前解决这些管理上的细节是必不可少的。

7.3　性能的扩展

Jackson 的程序优化规则。

- 规则 1：不要优化。
- 规则 2（仅针对专家）：还是不要优化。

——Michael A. Jackson，独立计算机顾问

程序员容易痴迷于性能、大 O 符号和可扩展性，会赞美一些“网络级”和处理“大数据”的公司。但现实却是，对于大多数创业公司而言，这些都不是特别重要的问题。我们的时间更应该花在能提高开发团队效率的工具和实践上，而不是让服务器跑得更快。事实上，做一些不能扩展的事情是早期创业成功的一个重要因素（阅读 3.2.4 节了解更多信息）。

只有在产品已经有了一定规模，性能可能成为瓶颈之后，这才是个需要考虑的问题。也就是说，如果我们足够幸运，遇到了这样的问题，这就是一个需要解决的重要问题。一旦网页加载的时间超过 3 秒，会有 40% 的用户放弃，对于网上购物的人群该数字上升到 57%，对于 18~24 岁的人群该数字上升到 65%。最糟糕的是，体验到网站速度缓慢的访问者中有 79% 可能不会再回来。

一般来说，考虑性能或软件开发的基本过程就是让它工作，让它正确、让它快速[7]。这是几个连续的步骤，我们必须按顺序去实现。实现错误功能的软件，即便它非常有效率也没有什么价值，所以我们在担心性能问题之前，必须先担心正确性的问题。只有得到整洁、可靠的代码之后，性能调优才是我们应该开始考虑的事情。性能的改进是以下两个步骤的迭代过程：

(1) 测量；
(2) 优化。

7.3.1 测量

下面是一个对计算机科学基础知识的有趣测试，出自 C++ 语言之父 Bjarne Stroustrup。我将生成 N 个随机整数，而你需要把它们按照排序插入一个列表中。举个例子，如果我生成数字 5、1、4、2，你的列表应该如下：

```
- []            // 初始列表
- [5]           // 添加5
- [1 5]         // 添加1
- [1 4 5]       // 添加4
- [1 2 4 5]     // 添加2
```

现在我将生成 0 和列表长度之间的随机索引，你需要将该索引对应的元素从列表中去掉。举个例子，如果我生成的索引是 1、2、0、0，你的列表应该是：

```
- [1 2 4 5]   // 初始列表
- [1 4 5]     // 去掉索引1
- [1 4]       // 去掉索引2
- [4]         // 去掉索引0
- []          // 去掉索引0
```

问题就是，当 N 是什么值的时候，使用链表去存储这一序列比使用数组更有效率？花一分钟的时间去考虑考虑。

回顾一下计算机科学的知识，你肯定知道链表应该更有效率，特别是当 N 越来越大的时候。因为数组的随机插入和删除可能需要改变整个数组的大小，而在链表中，更新两个指针是一种时间固定的操作。不幸的是，这个显而易见的答案却是错误的。如果你在 C++ 中对代码进行实际的性能测试，会发现采用数组的方法会快出几个数量级。现在，也许你会想到这是因为数组可以进行时间固定的索引查询，所以它可以对插入进行二分查找（O(log N)，而不是链表的 O(N)），并且可以对删除进行直接查询（O(1)，而不是链表的 O(N)）。好吧，我们将数组的实现改成对插入和删除进行线性扫描，在条件更加同等的情况下对比。现在你认为哪个方案会更快呢？

注 7：通常认为这种说法来自 Kent Beck，尽管很难找到确切的来源。

计算机科学的知识会再次清楚地告诉你肯定是链表更快。同样，这个显而易见的答案又错了。其实在所有情况下，数组的方法都要比链表的方法快 50~100 倍。原因是线性搜索在插入和删除的时间中占支配地位，尽管在大 O 分析中并不会表现出来，但数组中的线性搜索要比链表快得多。为什么呢？这是缓存一致性引起的。数组中所有的元素在内存中都是连续的，对它们进行线性搜索是一种可预测的访问模式，可以在 CPU 的一级和二级缓存中有效地缓存起来。另一方面，对于链表来说，它并不是一种连续的数据结构，每次跳到下一个或前一个指针都是一种随机的访问，通常都无法命中缓存，所以只能够跳到主内存中，速度大概慢了 50~100 倍。更糟糕的是，链表的内存开销大概是数组的 4 倍，因为它需要存放每一个元素的前后指针，所以要从内存中读取更多的数据。

这个练习的目的不在于说服你永远不应该使用链表，而是要让你意识到，没有测量是无法对代码的性能做出预判的。即便像列表插入和删除这样简单的操作，你的直觉和在学校里学到的大 O 符号几乎都会让你陷入迷途。

> 在没有数据的情况下，任何程序员都永远无法对性能瓶颈的位置进行预判或分析。不管你认为瓶颈会出现在哪里，都将惊讶地发现它出现在别的地方。
>
> ——Steve Mcconnell，《代码大全》

因此，性能调优的第一个步骤永远都是测量。我们必须用工具对代码进行测量，既要使用监控工具（阅读 8.5 节了解更多信息），也要使用性能工具和分析工具。在你收集到能够识别出最大瓶颈的确切数据之前，请不要进行任何的性能优化。在大多数情况下，你会发现几个存在潜在风险的点，通常完全是在意料之外的，是大多数性能开销的原因所在。举个例子，大部分创业公司最常见的瓶颈并不在于编程语言、Web 框架或任何算法的性能，而在于 I/O，比如远程 Web 服务调用或者从硬盘上读取数据。这是因为 I/O 的速度要比内存中或者 CPU 的操作慢几个数量级，所以即便很少被大 O 分析捕捉到，I/O 还是对性能有主要的影响。这些就是我们要通过性能分析去找出的瓶颈，也是需要花全部时间去优化的问题。

7.3.2 优化

编写高性能代码或实现可扩展系统的技术与特定的用户场景有极大的关系。我可以给大家提供一份改进性能和可扩展性的最常见的高级策略列表。

分治

　我们可以把一个问题划分成许多更小的问题，并在许多 CPU 和服务器上分别去解决这些问题，这样每一部分需要完成的任务会更少，例如多台 Web 服务器、数据库的复制或分区以及 MapReduce。

缓存

　我们可以提前执行任务并保存结果，这样就不用在需要的时候再计算，只需要从存储中获取提前计算好的值即可，例如数据库缓存、非规范化架构、分布式缓存、CDN、cookies、记忆化（memoization）和动态编程算法。

懒惰

　除非绝对必要，否则可以先推迟一些任务，之后再做，例如延迟加载网页的部分内容、只在滚动到该区域之后再加载、数据库中的乐观锁定。

近似正确性

在许多情况下，得到一个“足够接近”的答案比得到准确的答案能减少很多工作量，例如最终一致性、HyperLogLog、降低持久性保证、尽力消息传递（best-effort messaging）。

异步

在等待计算结果的时候不采用锁定或阻塞的方法，而是继续执行任务，在计算完成的时候再通知你，例如非阻塞 I/O、事件循环、无锁数据结构。

抖动与随机化

尝试将负载均匀扩散开，从而避免出现峰值和热点，例如缓存有效期限随机化、负载均衡算法（轮循环与优先级调度）、键分区算法（范围分区与哈希分区）。

节流

拒绝某些计算，以防拖慢其他计算，例如服务器限速请求或去除请求路径中的缓慢服务器。

冗余

剔除一次以上的相同计算并返回最快完成的计算，例如分布式系统中的备份或多面请求（hedged requests）、应对故障的冗余服务器（数据库双机热备）。

协同定位

把物理上更接近的东西放到一起以降低延迟，例如 CDN、全世界的多数据中心、将关联的服务器放在同一机架上。

更快的硬件

又称为垂直扩展（vertical scaling），例如更快的 CPU、更多的内存、更多的 CPU 缓存、固态硬盘、更快的网络、在内存中执行计算而不是在磁盘上计算，或者用在 CPU 缓存中计算代替在内存中计算。

更快的算法

找到可以减少工作量的算法，例如用二分搜索代替线性搜索、用快速排序代替冒泡排序。

在实现这些策略之前，我们可以先进行一些估算，判断哪种策略对我们可能最为划算。当然，也只有实际实现和测量之后才知道确切的结果，但是实现的成本是昂贵的。我们可以使用基本的算术，排除一些明显不好的选择，节省许多时间。为此，我们需要知道所有各主要系统的关键指标，比如数据库的主键查询要花多长时间，数据中心一次请求的来回时间。（再次强调，提前测量是最重要的！）我们可以查看“每名程序员都应该知道的延迟数字”得到大概的数值。

举例来说，假设我们正在实现一个搜索应用程序。搜索索引增长得太大，单台服务器的内存已经容纳不下，我们就可以选择把它存放在磁盘上或者将其划分到 10 台服务器上，每台服务器在内存中存放索引的 1/10。那么，是在硬盘上进行本地查询更快，还是并行地扇出到 10 个服务，每次都从内存中查询更快呢？假设我们在处理一个典型的搜索查询，必须连续地读取 1MB 的数据并在计算中使用以下数值：

- 主内存引用耗费大约 100ns；
- 从内存中连续读取 1MB 数据耗费大约 12 000ns；

- 随机磁盘检索耗费大约 4 000 000ns；
- 从磁盘连续读取 1MB 数据大约耗费 2 000 000ns；
- 在同一数据中心的一次来回耗费 500 000ns。

有了这些数字，我们在本地硬盘上查询数据所花的时间为：

```
延迟 = 1次磁盘检索 + 读取1MB数据
延迟 = 4 000 000 ns + 2 000 000 ns
延迟 = 6 000 000 ns
```

并行地扇出到 10 个服务，在内存中查询数据耗费的时间是：

```
延迟 = 1次数据中心来回 + 1次内存引用 + 读取1MB数据
延迟 = 500,000 ns + 100 ns + 12,000 ns
延迟 = 512,100 ns
```

看起来把代码分解到一堆服务上似乎会更快，但是先等等，如果使用 SSD 硬盘会如何呢？我们看看它们的数值：

- SSD 的随机检索耗费 16 000ns；
- 从 SSD 连续读取 1MB 数据耗费 200 000ns。

对于 SSD，处理本地查询将耗费：

```
延迟 = 1次SSD检索 + 读取1MB数据
延迟 = 16,000 ns + 200,000 ns
延迟 = 216,000 ns
```

这样看来，升级到 SSD 也许可以让我们得到最佳的性能。SSD 虽然要花钱，但如果能让你免于重新编写所有代码去使用远程服务，总体上可能成本更低。因为就目前来说，程序员的时间大概比 CPU 的时间要昂贵三个数量级以上。在大多数情况下，如果我们可以通过购买或租用更快的硬件来解决问题，将比通过重写代码解决问题更加便宜。

7.4 小结

从某种程度上看，编程就像力量训练。力量训练的初学者会发现他们所做的几乎一切训练都将产生某些结果。你走进一个健身房，做几个二头弯举，你的手臂就会开始变粗；每天跑上几公里，你的腰围就会变细。这种效果只会持续到某一个时刻。之后的一段时间，也许是一年，你就会遇到瓶颈，所有的过程似乎都停止了。你不断举重，但是并没有变得更强壮；你不断奔跑，也没有变得更瘦。此时继续前进的唯一方式就是开始使用更高级的训练计划，你要懂得调整哑铃的重量，需要吃更多的蛋白质，需要更多的睡眠，你要懂得某些练习比其他的练习更有效果。突破瓶颈的唯一方式就是从根本上改变你的训练。

其实编程也是一样的。

> 我的朋友 Clift Norris 找到了一个基本的常数，我称之为 Norris 常数，表示未经训练的程序员在他（或者她）遇到瓶颈之前所能编写的平均代码量。Clift 估计这个数是 1500 行。如果代码超过这个数，就会十分混乱，作者也无法轻松地调试或修改。
>
> ——John D. Cook，*The Endeavour*

就像在举重一样，你几乎可以在编程领域做任何事情——复制与粘贴、使用全局变量、不编写测试和文档——也能够工作。不用多长时间，最终，你就会遇到天花板。如果想要突破它，也一样需要更高级的方法，即要遵循本书所描述的整洁的编码原则和实践。遇到天花板的时间也不会太长：仅仅生成一个全新的 Ruby on Rails 应用的框架就有大约 900 行代码，已经超过了 Norris 常数的一半。

事实上，这样的瓶颈不仅只有这一处。Norris 之数代表了编写一两千行代码就会遇到的问题。为了克服这些问题，我们必须开始考虑上一章所提到的一些编写整洁代码的原则，比如更好地命名、代码布局、松耦合和高内聚。等到了大概 2 万行代码的时候，我们又会遇到另一个瓶颈。

> 我在毕业后的第一份工作中，就重复地遇到了 2 万行代码的瓶颈，我的同事也遇到同样的情况（跟我一样年轻）。在 DreamWorks，我们有 950 个程序供动画师使用，代码计数器表明大一点的程序大概在 2 万到 2.5 万行代码左右。如果超过这一数字，就很难轻易地添加新的特性。
>
> ——Lawrence Kesteloot，*Team Ten*

为了克服这一瓶颈，我们就必须在编码实践上多想想办法，可能要多花些时间进行代码评审、自动化测试、重构、将代码分解成更小的函数和模块、控制代码的副作用。这么做将使我们走得更远，直至达到几十万或几百万行代码才会遇到新的瓶颈。

> 似乎在三四百万行代码的时候，就会遇到瓶颈。在达到 300 万行代码之后，不管多少人（数以百计）花多少年（数以十年计）参与进去，增长率似乎都会明显变慢。
>
> 许多产品的公司专有代码量大致都是以百万行计的，尽管其中也包含了大量的死代码。NVIDIA 的核心驱动代码正好是 300 万行，尽管附属功能的代码要多出 100 万 ~ 1000 万行。游戏厂商的代码似乎会少一点，在 150 万 ~200 万的范围内，可能也是因为它们应用程序的数量更少一些。
>
> ——Daniel Wexler，Wexworks 创始人

对于拥有几十万或几百万行代码的代码库来说，需要的是完全不同的实践：独立的代码存储库、严格控制和向后兼容的 API、大量的单元测试、全面完整的文档，等等。

所以，我们又要回到创业公司是与人密不可分这个事实上。我们对代码库的维护能力和我们所使用的技术并没有那么大的关系，更多是与人类大脑和心理的天生局限性有关。如果我们主要是自己编写 2000 行以下的临时程序，比如一开始创业时多半要做的抛弃型的原型，那么不用考虑太多可扩展的编码实践就可以开干了。但是如果要编写规模更大、有更多人参与、需要处理更多负载的东西，从根本上就是完全不同的做法。我们必须从一开始就关注所有可以帮助人们阅读和编写代码的事情，比如命名、代码布局、内聚、耦合、重构、测试、代码评审和文档。如果能这样做，就会前进得更快，速度制胜。

> 我们在 Peak Strategy 的时候遇到过一个情况：我们让一位客户第二天来公司，他本应该是在第二天早 9:00 或 9:30 到的。他在一家期货及衍生品交易公司工作，他的公司具有非常高的交易速度，和我们以前处理的交易类型有很大不同。我们需要为他做些演示，因为我们的软件并不能完成这样的任务。我不记得确切的情况，但是不知怎的我们被逼到了墙角。

在那天下午早些时候，也就是在他本应该出现的时间之前，我们正兵分三路进行结对编程，实现第二天的 demo 所缺失的系统模块。那时我们都对纯粹的方法论充满渴望，所以我们约定将使用纯粹的测试驱动开发、结对编程和细致的重构，等等。我们从中午就一直工作到第二天早上 7 点。我们的工作效率实在是了不起，而且据我所知一点问题都没有。我想说的是，在那样的环境中出现错误是非常可怕的，让一个金融工程师在工具执行的计算中找出一个 bug 是极其可怕的。但是我们取得了难以置信的进展，却没有引入任何的 bug，也在第二天有了一个完美无瑕的演示。

当你开始尝试在软件公司中引入过程方法的时候，总会听到这样的声音："我只能在不赶时间的时候去使用好的方法。如果时间很赶，我就管不了那么多了"。我总喜欢把这个故事拿出来举例子，想告诉大家一个真正好的方法无论在什么尺度上都应该能够加快你的速度。一个好方法可以缩短你一小时、一天或一年的时间。

——Dean Thompson，NOWAIT 公司 CTO

第 8 章

软件交付

8.1 完成意味着交付

软件还在电脑上运行并不算完成，代码整洁并通过了测试也不算完成，别人在代码评审时告诉你可以“发布”也不算完成，当你把便利贴拖到“功能完成”一栏也不算完成，只有把软件**交付**给用户之后，软件才算完成了。

许多公司使用的是手动交付过程。我访谈过一些创业公司，他们的开发人员会来回地用 email 发送代码，通过 FTP 或 SSH 手动方式将新代码上传到服务器上，手工去配置每一台服务器，整个公司中只有一名工程师知道让一切运转的魔法咒语。你也会在 8.2 节看到，这种点对点的方法可能会给创业公司带来许许多多的痛苦。

> 如果你觉得痛苦，就多去做。
>
> ——Martin Fowler，程序员、作者、Thoughtworks 讲师

本章经常介绍的一个主题就是，降低交付过程痛苦的最佳方法与我们的直觉相违背，那就是要更加频繁地去面对这一痛苦。为此，我们需要在创业公司中建立起交付过程，处理编写完代码**之后**伴随而来的各个关键步骤。这些步骤分别是构建、部署和监控。这一章将向你展示如何以一种快速、可靠和自动化的方式去应对这些步骤。

8.2 手工交付：一个恐怖的故事

2011 年，LinkedIn 上市了，估值在 100 亿美元。它的会员在以每秒 2 个的速度增长，但它的交付过程存在很大问题，导致公司几乎停滞不前。这是在几乎所有超速发展的公司中都存在的普遍问题，但迄今为止，很少公司有足够的勇气把情况说出来。

我们使用了一个火车发布模型。在这个模型中，每两周，一列“火车”就会带着新的代码离开车站，投入生产。为了搭上火车，你需要让代码进入发布分支。那时，团队在独

立的功能分支上进行自己的任务，各个团队间在数周或数月的时间里是完全隔离工作的。接着，在安排的发布日期前三周，所有功能分支的开发都会暂停下来，十几个团队会尝试将修改合并到发布分支中。

合并的过程就是一个噩梦，开发人员几个月来编码所依据的假设已经不再有效。你所使用的类已不复存在，数据库架构也发生了变化，在几十个地方使用的 API 已经被重构了，UI 看起来也完全不同，你认为最终已经从代码库中拿掉的 JavaScript 库现在又被用在十个新的地方。这些冲突要花好多天去解决，而在完成之后，你又意识到合并过程中有这么多的代码已经改变了，自己在功能分支上所做的大部分测试也变得没有价值了。

我们的测试还远远不够。我们进行了测试，但只对一部分代码进行了测试，仍然存在两个问题。第一，这些测试太慢了，真的很慢。构建的过程——编译、运行测试和封装，所花的时间是以 10 小时来计的。第二，这些测试并不可靠。其中有许多测试很诡异，会间歇出现失败。所以每次合并之后，大致每天都要构建一次，看看是否能正常工作。我们面对的是几十个测试失败，无从了解这些失败是由某个功能分支中的 bug、有问题的合并，还是仅仅是测试出现莫名其妙的情况引起的。

假设我们能够使构建过程变得稳定，接下来的挑战就是整理汇集出**部署计划**。这是一个手工维护的 wiki 页面，它会列出发布时需要部署的所有服务，以及这些服务要按照什么顺序部署，需要进行什么配置。举例来说，假设你在功能分支上修改了 B 服务，为它添加了一个新的端点（endpoint），又修改了 A 服务调用这个新的端点，你就必须记住要把 A 和 B 这两个服务都添加到部署计划中，并且要指明 B 必须先行部署，这样新的端点才可供 A 使用。如果你需要修改 B 服务的配置，比如增加其线程池的大小或者调整垃圾回收设置，也必须记得把这些修改放到部署计划中，因为所有的配置都是手工管理的。

部署剩下的工作也是手工进行的，我们的发布团队会浏览 wiki 网页，按部就班地在几千台服务器上，对数百个服务进行部署并修改配置，要么通过手工进行，要么使用一些临时的 shell 脚本。由于这一切都是手工操作，不可避免会犯下错误。有一些错误是很明显的，有一些则需要好几个小时或好多天才会发现，因为我们只对少数位置进行监控。大多数情况下，我们都是盲目行动，直到有用户抱怨了，才知道有地方出错了。有些东西会一直出错，所以发布要花好几个小时，一直拖延到夜里。有些发布要花好几天才能完成，少数发布不得不半途取消，因为我们无法让代码保持稳定。

到了 2011 年年底，我们已经快到了根本无法发布代码的境地，我们开始进行项目反转，彻底对发布过程进行检查，等到完成的时候[1]，我们已经能够做到在一个小时内进行多次部署，而 bug 的数量要少得多，并且拥有全面的监控解决方案去捕捉漏过的任何错误。我们是如何实现这样激动人心的改进的呢？第一个步骤就是改进构建的过程。

8.3 构建

构建过程由三个部分构成：版本控制、构建工具以及一个将它们集中起来持续集成过程。

注 1：实际上这是一个增量式的、持续的过程，永远不会真正地完成，但我们每隔几个月就能够看到巨大的改变。

8.3.1　版本控制

版本控制系统（version control system，VCS）可以让你对一组文件的变化进行持续跟踪。即便你从来没有使用过 VCS，可能也有好多次可以东拼西凑出自己的系统来。你是否曾把一个文件的副本通过 email 发给自己作为备份？是否和同事间用 DropBox 共享文档？你的硬盘上是不是存在 15 个版本的简历（resume-v1.doc、resume-v2.doc、resume-09-03-14.doc）？这些临时办法对几个 Word 文档或电子表格来说也许没问题，但并不是管理软件的好方法。

如果你正在从事编码工作，却还没有找到好的解决方法，那么你需要试试 VCS 了。没有什么正当理由可以不去使用 VCS，一个都没有——即便你在做一个单人的项目，即便只是一个小项目，即便这个项目你开发了几年都没有用 VCS。另外，设置一个 VCS 的成本也是非常低的，而它带来的好处却很多。现在你是没有任何借口了。通过存储每个文件完整的修改历史，VCS 可以给你和你的团队带来超能力。如果你犯了错，可以恢复到任何文件的最近版本。如果你发现了 bug，可以挖掘近期的**提交**和**提交信息**去追踪引发这个 bug 的修改。如果你同时开发多个功能，可以把它们放在独立的**分支**上。如果你需要和团队成员合作，可以基于**提交钩子**、**合并**和 **pull 请求**去实现一个工作流。

到 2015 年为止，最流行的版本控制系统就是 SVN 和 Git[2]。SVN 是一个**集中式的 VCS**，信任源是一个中心服务器上的版本库，所有开发人员使用客户端软件和该库进行交互，如图 8-1 所示。要获得代码的副本，我们要将其从中心服务器上**签出**。如果修改了其中的部分代码，要将这些修改**提交**回中心服务器上。

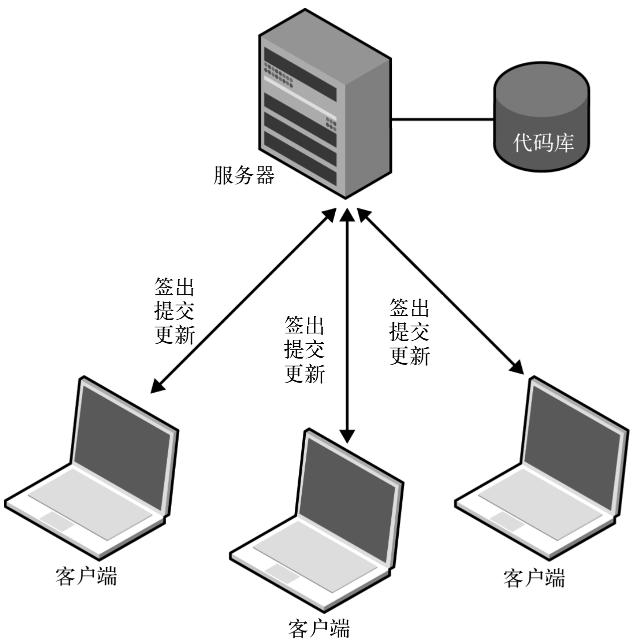


图 8-1：集中式版本控制

注 2：请访问 StackOverlow 的开发人员调查，了解相关数字。

Git 是一个**分布式的 VCS**，每个人都有一份代码库的副本，同时起到服务器和客户端的作用，如图 8-2 所示。我们可以**克隆**别人的代码库来获得代码的副本，可以把修改**提交**到本地的代码库，可以把最新的提交通过**推送**提交给其他人，也可以通过**拉取**获得他人最近的提交。

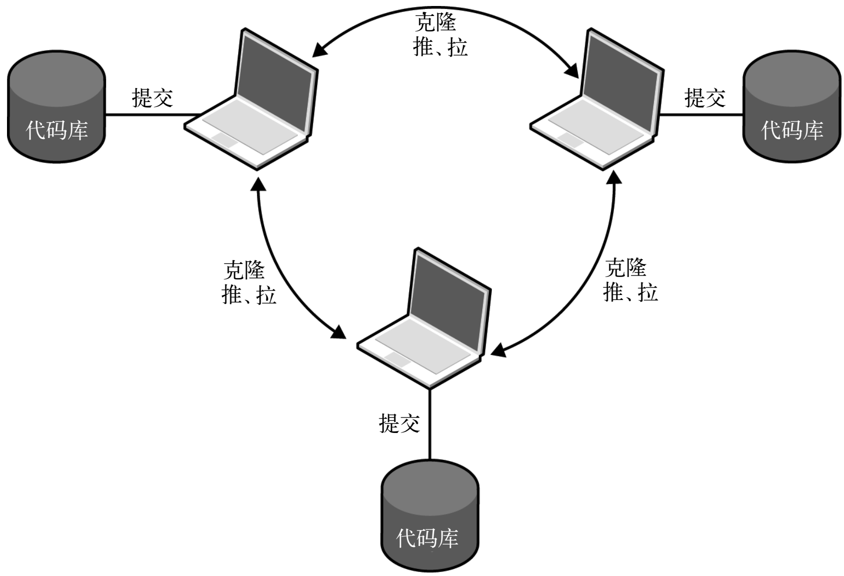


图 8-2：分布式版本控制

在实际中，即便是分布式的 VCS，通常也有一个中心节点作为正式的信任源，用于所有的构建和发布。例如，GitHub 就是大部分开源项目的信任源，我们必须把提交推送到上面，使它们成为“正式的”。

那么是应该在创业公司使用集中式的 VCS，还是分布式的 VCS 呢？对于大多数公司来说，Git 可能是最好的选择，部分原因是因为它是分布式的，部分原因则因为它正开始在版本控制领域占据统治地位，这是由于 GitHub 和开源软件的流行。也可以这么说，任何 VCS 都比没有要好，所以如果你更喜欢其他工具，使用它也完全没问题。无论你选择的是哪一种 VCS，都应该遵循两个最佳实践：编写良好的提交信息、及早提交并经常提交。

1. 编写良好的提交信息

假设我们部署新版网站到生产环境中时，发现搜索功能出了问题。为了找出原因，我们打开提交日志，浏览最近的发布做了什么改变：

```
> git log --pretty=oneline --abbrev-commit
e456b8b Maybe this will do it
d98846a Fix another issue
59635e9 Fix stuff
964ce4c Initial commit
```

你能猜到这些提交哪一次把搜索功能弄出问题了吗？不行？好吧，如果看到的是下面的情况呢？下面是同样的提交，但描述是不同的：

```
> git log --pretty=oneline --abbrev-commit
e456b8b Add alt text to all images
d98846a Improve search performance by adding a cache
59635e9 Update logo on homepage
964ce4c Initial commit
```

现在可以清楚地看到编号 d98846a 的提交“添加缓存以改进搜索性能”是最有可能引发这个 bug 的。我们可以查看该提交的**差异**，准确了解修改了什么，这将节省调试的时间。所有的 VCS 都允许我们在签入代码的时候提供**提交信息**。如果我们能够正确使用这一功能，它们就会成为跟踪 bug 和尝试弄清楚代码变化情况的关键来源。这就是我们为什么应该花时间编写好的提交信息。

良好的提交信息由概要和描述构成。概要就像论文的标题：单独位于第一行，应该简短而切中要点[3]。在概要之后，应该插入一个新行，以段落或要点的格式解释我们修改了什么、为什么要修改、哪里可以找到更多信息（例如指向 bug 跟踪器、wiki 或代码评审的链接）。下面是一个例子：

```
Improve search performance by adding a cache

To improve the latency of our search page, we are now using
memcached to cache search results for a configurable
amount of time (default is 10 minutes). Results from
memcached come back in 1ms instead of the 100ms it takes
to hit the search cluster.

- Full design: http://wiki.mycompany.com/search-caching
- JIRA ticket: http://jira.mycompany.com/12345
- Code review: http://reviewboard.mycompany.com/67890
```

2. 及早提交并经常提交

如果我们长时间没有提交，有地方出现了错误，VCS 就无法提供帮助。因此，我们应该及早提交并经常提交。如果出现了问题，这些提交就能像检查点那样，让我们更容易地跟踪到原因。在必要的情况下，还可以向前恢复几步。理想状态就是每一次提交都是**完全实现**了某一个**单一目的**、**大小合理的单元**。我们把这句话拆开看看。

单一目的意味着我们不应该在同一次提交中修复两个 bug 或者实现两个功能，或者在一次提交中重构现有代码和实现新的代码。**完全实现**意味着我们不应该提交会给构建过程带来问题的代码，或者让用户看到未完成功能的代码。**大小合理的单元**意味着我们应该把工作分解成较小的、增量式的步骤。并非巧合，这也恰好就是测试驱动开发、重构和代码评审（阅读第 7 章了解更多信息）成功的秘密。例如，如果我们开发的功能要花好几天才能实现，也许就应该把它分成三次提交：一次提交可能添加一些失败的测试用例（先标记为忽略，这样构建过程就不会失败）；第二次提交可能对现有代码进行重构，以便可以轻松实现新功能；最后一次提交则实现实际的功能。读者可以阅读 8.3.3 节，了解如何将大的功能分解为较小的、安全的提交。

注 3：概要通常应该少于 50 个字，才能在工具（比如 `git log` 命令）中很好地显示出来。

8.3.2　构建工具

每一个代码库都需要构建工具对其进行编译、运行测试，以及封装代码成用户产品。目前有许多开源的构建工具可供使用，具体使用哪一种取决于我们所要编译的代码类型。例如，如果你正在用 Ruby 开发，可能就要使用 Rake；如果你正在用 Scala，可能要用 SBT；如果你正在用许多不同的编程语言，Gradle 也许是最佳选择；如果你正在编译静态内容，Grunt.js 和 Gulp 提供了大量插件可以应对各种常见任务，比如连接 CSS 和 JavaScript 或者减小它们的尺寸，或对 CoffeeScript、Sass 和 Less 进行预处理。

大部分构建系统也都可以帮助我们管理依赖项。如果代码依赖于第三方库或开源库，我们就不应该只是把依赖项的代码直接复制并粘贴到项目中。假如这么做的话，也得把传递依赖项的整棵树的代码复制、粘贴进去。举个例子，如果你依赖 A 库，A 又依赖 B 和 C，C 又依赖 D、E 和 F，就必须把所有这些库的代码都复制到项目中。当你想要升级到新版的 A 库时，也许会发现它正在使用新版的 B 和 C，还加入了新的依赖项 C，这时你就不得不也去升级所有这些依赖树。

这样下去很快就没办法管理了，所以大部分的构建系统都可以让我们制定顶层的依赖项，它们会负责为你传递依赖项。例如，下面的代码是在 Gradle 中制定依赖项：

```
dependencies {
  compile group: 'commons-io', name: 'commons-io', version: '2.4'
  testCompile group: 'junit', name: 'junit', version: '4.+'
}

repositories {
  mavenCentral()
}
```

这段代码将告诉 Gradle，我们需要版本为 2.4 的 `commons-io` 库才能编译代码，还需要版本为 4.0 或 4.0 以上的 `junit` 库去编译测试代码，Gradle 可以在 Maven Central 代码库中找到这些库。所以，当我们编译或运行代码时，Gradle 将会自动下载这些库，加入它们的所有传递依赖项，把它们包含在类路径中。

8.3.3　持续集成

假设你正在负责建立国际空间站（International Space Station，ISS ），它是由几十个组件构成的，如图 8-3 所示。

每个组件都由独立的团队实现，组织的方式取决于你，你有两个选择。

(1) 提前做出所有组件的设计，然后让每个团队单独完成各自的组件，直到完成为止。当所有团队都完成的时候，把所有组件发射到太空，然后尝试同时将它们组合起来。

(2) 做出所有组件的初步设计，然后让每个团队去投入工作。在他们推进的过程中，不断对每个组件和其他所有组件进行测试，如果有问题则对设计进行更新。当组件完成后，把它们同时发射到太空中，增量式地组装起来。

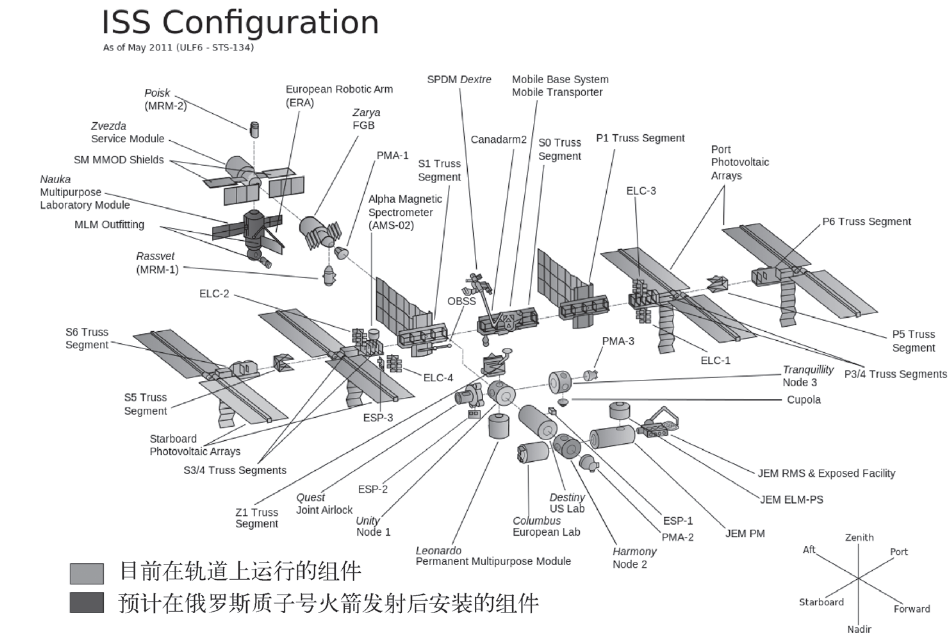


图 8-3：国际空间站

对于第一个选择，在最后一分钟尝试组装整个 ISS 将会暴露出大量冲突和设计问题。A 团队认为 B 团队会处理线路连接，而 B 团队认为 A 团队会处理；所有的团队都使用公制，只有一个除外，没有一个团队记得要安装马桶。不幸的是，因为所有东西都已经完全做好了并在太空中漂浮，把它们弄回来再修复将会非常昂贵和困难。很明显，这一选择将会是一个灾难，但是这正是许多公司做软件的方法。开发人员同时完全隔离工作数周或数月，到了最后一分钟又尝试将所有的代码合并到一起。这个过程就是所谓的**后期集成**，就像我们在本章开头看到的 LinkedIn 的故事一样，通常都会导致灾难。

更好的方法就是第二种选择所描述的，是**持续集成**，所有开发人员定期（每天或者每天多次）将代码合并到一起，这一过程可以尽早暴露设计中的问题，避免在错误的方向上走得太远，我们也可以增量式地改进设计。实现持续集成最常见的方法就是使用**基于主干的开发模型**。

1. 基于主干的开发

在基于主干的开发模型中，开发人员在同一个分支（通常是 `trunk`、`HEAD` 或 `master`，取决于你的 VCS 怎么命名）上进行他们的工作。这里不存在功能分支[4]。看起来似乎让所有开发人员在单一的分支上工作是不可能扩展的，但现实是这种方法也是扩展的唯一方法 。LinkedIn 去掉功能分支，并把基于主干的开发作为实现项目反转的一部分，也是现实从 100 名左右的开发人员扩展到超过 500 名开发人员的必备条件。Facebook 使用基于主干的开发很好地实现了超过 1000 名开发人员的扩展。Google 使用这种模型已有多年，表明基于主

注 4：即不使用功能分支在开发人员之间共享代码，开发人员仍然可以在他们的本地环境中使用分支同步开发多个功能，但在他们准备和他人分享这些修改的时候，总是要把代码推送回主干。

干的开发可以支持 15 000 名以上开发人员、4000 个以上的项目和每分钟 20~60 次提交。

成千上万的开发人员如何才能做到频繁地签入同一分支却没有冲突呢？事实证明，如果你进行小的、频繁的提交，而不是规模庞大的提交，冲突的数量就会相当小，而这样的一些冲突也是可接受的。这是因为不管使用什么集成策略，处理只需一两天工作（使用持续集成）造成的冲突要比处理需要几个月的工作（使用后期集成）造成的冲突要更容易一些。

基于主干的开发让冲突问题没有那么严重，但是稳定性又如何呢？如果所有开发人员都在同一分支上工作，而一名开发人员签入了无法编译或引起严重 bug 的代码，可能就会阻碍所有的开发。为了防止这种情况发生，我们必须使用**自测试构建**。自测试构建是一种完全自动化的构建过程（即可以在单条命令中运行），该过程具有足够的自动化测试。如果测试均通过，可以确信代码是稳定的（阅读 7.2.1 节了解更多信息）。常用的方法是添加一个提交钩子到 VCS 中，每一次提交均在持续集成服务器上（CI 服务器，比如 Jenkins 或 Travis）进行构建，如果构建失败则拒绝该提交[5]。CI 服务器是你的守护者，它在允许代码进入主干之前会对每一次签入进行验证。

> 如果没有持续集成，在有人证明你的软件可行之前，可以说它都是有问题的，而通常得等到测试或集成阶段才能被证明。对于持续集成，你的软件所出现的每一次新变化都被证明是可行的（假设具有非常全面的自动化测试）——你也知道它什么时候出现问题，可以立即把问题修复。
>
> ——Jez Humble 和 David Farley，《持续交付》

持续集成对于小规模、频繁的提交是很有好处的，但如何应对大的修改呢？如果你开发的是耗费数周或数月的东西，如何才能定期提交未完成的工作，又不至于破坏构建过程或不小心发布未完成的功能给用户呢？答案就是使用**抽象分支**和**功能开关**。

2. 抽象分支

解释**抽象分支**最简单的方法就是使用一个例子。我们假设正在 Java 应用程序中使用 Redis 键 - 值存储，并使用 Jedis 客户端库去访问它：

```
Jedis jedis = new Jedis("localhost");
String value = jedis.get("foo");
```

我们要用 Voldemort 的键 - 值存储去代替 Redis，但整个代码库中有数以千计的地方都在用 Jedis 库。有一种选择就是创建一个功能分支，然后就可以花几个星期将所有客户隔离开，修改为使用 Voldemort 库，然后再寄希望于可以安全地把代码合并回去。要实现同样的隔离，更好的办法是在代码中使用抽象。举例来说，可以先为键 - 值存储定义一个简单的接口：

```
public interface KeyValueStore {
  String get(String key);
}
```

然后创建该接口的实现，在底层使用 Jedis：

注 5：在等待代码评审的时候运行构建是一种不错的优化方法，到评审员看到代码时，通常已经知道这次签入是否会引起问题。

```
public class JedisKeyValueStore implements KeyValueStore {
  private final Jedis jedis = new Jedis("localhost");

  @Override
  public String get(String key) {
    return jedis.get(key);
  }
}
```

可以在主干中直接实现并测试这个新的类。因为没有人使用它，所以可以很轻松地通过几次小的提交来实现，不会把构建过程弄出问题。准备好这个类之后，就可以开始迁移所有使用 Jedis 客户端的代码，让它们使用我们的抽象：

```
KeyValueStore store = new JedisKeyValueStore();
String value = store.get("foo");
```

由于这一改变并不影响任何外部的行为，我们可以增量式地修改客户端，期间可以进行多次小的签入。与此同时，我们也可以实现并测试底层使用了 Voldemort 的 `KeyValueStore` 抽象的新的实现：

```
public class VoldemortKeyValueStore implements KeyValueStore {
  private final StoreClient<String, String> client =
    new SocketStoreClientFactory(
      new ClientConfig().setBootstrapUrls("tcp://localhost:6666")
    ).getStoreClient("my_store_name");

  @Override
  public String get(String key) {
    return client.getValue(key);
  }
}
```

同样，由于没有人使用该实现，我们可以直接在主干中进行构建和测试，进行多次小的签入。准备好之后，所有客户端都已经被迁移到这个抽象中，我们可以开始迁移它们去使用新的 Voldemort 实现：

```
KeyValueStore store = new VoldemortKeyValueStore();
String value = store.get("foo");
```

还是一样，可以在主干中增量式地进行这种修改。事实上，对于新的键－值存储，一次测试一个用例可能比较合适，我们可以在完成整个产品的迁移之前找到 bug。最终，我们将完成所有客户端的迁移，安全地从代码库中去掉 Jedis 抽象。

值得注意的是，抽象分支实际上仅仅是依赖反转原则的实践（阅读 6.8 节了解更多信息），它不仅仅是在没有功能分支的情况下进行重大的重构，在很多情况下，它还可以产生更整洁的代码。

3. 功能开关

功能开关背后的思路是未完成或有风险的代码在默认情况下应该是不可用的，在它完成的时候应该有简单的方法去启用它。这种方式可以让你把大的功能分解为小的、增量式的部分，只要一稳定就将其签入，而不是等到完全完成。例如，当我们为网站首页实现一个新的大模块时，可以把这个模块放在一个 `if` 语句中：

```
private static final String NEW_HOMEPAGE_MODULE_TOGGLE_KEY =
  "showNewHomepageModule";

if (featureToggles.isEnabled(NEW_HOMEPAGE_MODULE_TOGGLE_KEY)) {
  // 在主页上显示模块
} else {
  // 不要显示新的模块
}
```

在上面的代码段中，`featureToggles` 类会查找键，比如 `showNewHomepageModule`，它要么放在应用程序的配置中，要么从远程服务获取（例如键 – 值存储）。所有功能开关的默认状态都是关闭的，所以只要代码编译并通过了现有的测试，我们就可以在新模块的代码完成之前将其提交到主干中，没有用户会看到它。当我们完成该功能之后，就可以在配置或远程服务中开启这个功能。

在 LinkedIn 中，我们用来实现功能开关的远程服务被称为 XLNT。它有一个 Web UI，可以让我们动态地决定哪些会员可以看到哪些功能。例如，我们可以用 `showNewHomepageModule` 这个键决定让它只对 LinkedIn 员工可见，或者只对法语会员可见，或者只对美国 1% 的会员可见，如图 8-4 所示。

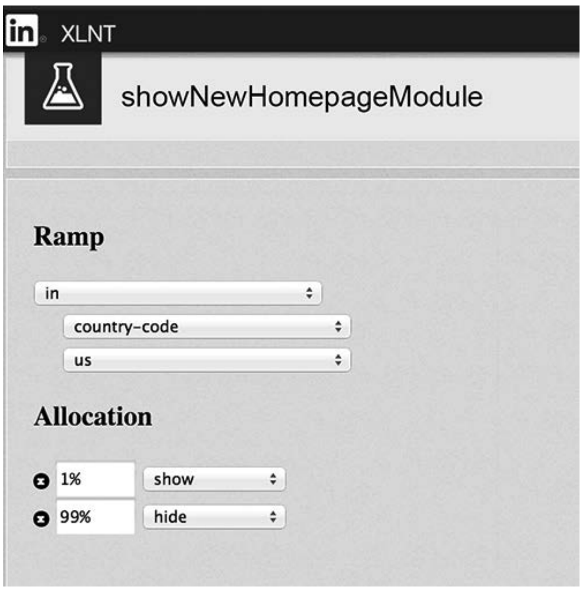


图 8-4：XLNT Web 界面

XLNT 不仅可以决定功能的开和关，还能够渐进式地让一个功能从完全关闭，到提供给 1%，到 10%，最后到 100% 的会员。在这其中的每个步骤中，如果我们发现了任何 bug 或者性能上的问题，都可以快速地让这个功能的覆盖率降下来。

当然，我们也不应该把所有东西都放到功能开关中[6]。功能开关与风险管理相关，如果风险较低，代码中遍布的 `if` 语句所带来的额外的复杂性也许就是不值得的。如果你使用了

注 6：实际上，有些东西即便你愿意，也不能轻易放在功能开关中，比如数据库模式的修改。

功能开关，必须在功能已经启用后严格清理干净，否则你的代码库将会被各种已经不再执行的代码分支弄得乱七八糟。至少，也要在每个功能开关的上方放一条 TODO 语句作为提醒[7]：

```
// TODO: remove this feature toggle after the ramp on 09/01/14.
// See http://mycompany.wiki.com/new-homepage-module for more info.
private static final String NEW_HOMEPAGE_MODULE_TOGGLE_KEY =
  "showNewHomepageModule";
```

8.4　部署

构建系统提供了一种可靠的方式，用于保存代码并进行测试和封装打包。要把封装好的代码部署到生产环境中，需要问四个问题。

- 部署到哪里？
- 部署什么？
- 怎样部署？
- 什么时候部署？

这四个问题的答案分别是：托管、配置管理、部署自动化和持续交付。

8.4.1　托管

每一间创业公司都需要在**自托管**和**云托管**之间做出选择。如果选择了自托管，就是把代码部署到自己拥有和管理的硬件之上，要么是放在自己的数据中心，要么是放在租用的他人的数据中心的机架上（也叫**主机托管**）；如果选择云托管，就要把代码部署在虚拟的服务器上，这些服务器运行在第三方拥有和管理的硬件上。

就像大部分创业公司应该使用开源软件而不是实现自己的基础框架一样（阅读 5.3 节了解更多信息），大部分创业公司，特别是在早期，也应该使用云托管而不是搭建自己的数据中心。购买和部署硬件会花费大量时间和先期成本，要做到可靠、冗余并持续地维护，甚至还要有更大的投入。大部分创业公司没有时间或金钱在内部专门培养在硬件、能源、制冷、网络和安全方面的专家。举个例子，2011 年和 2012 年 Y Combinator 创业公司只有不到 25% 采用了自托管。

采用主流提供商（像 Amazon、Rackspace、DigitalOcean 或 SoftLayer）提供的云托管可以获得更大的灵活性，能够以秒级提供服务器（例如响应流量的增长），而不是花数周去预定和安装自己的硬件。这些服务均提供了大型的社区，可以很容易地找到文档，学习最佳实践，利用开源插件和扩展，招聘到懂得如何使用它们的开发人员。大部分服务也都提供了自己的扩展，比如数据库管理、负载均衡、队列以及安全、部署、监控和分析工具。很多时候，云主机也是最便宜的选择，因为我们不用支付硬件的前期投入，只需根据使用的需要支付即可，只有在公司发展、变得更加成功之后，成本才会随之增长。

选择自托管唯一的原因就是对应用程序的性能有极高的要求。大部分云托管提供商提供

注 7：Ruby 和 Scala 均有 TODO 库可以指定一条具有过期期限的 TODO 语句。只要日期一过，该 TODO 语句就会引起构建失败。

的是运行在共享硬件之上的虚拟服务器。虚拟化的开销以及和其他用户竞争资源的需要，意味着需要从每台服务器获得最大 CPU 和硬件性能的应用程序也许需要运行在自己专用的服务器上。此外，许多云提供商会对数据存储和带宽收取额外的费用，所以需要处理大量图片、音乐或视频的创业公司可能会觉得云托管的价格过高了。

8.4.2 配置管理

在解决了“代码部署在哪里”的问题之后，我们需要解决“部署什么”的问题。尽管大部分开发人员考虑的只是应用程序的代码，但这只是必须安装在每台服务器上的长长的列表上的最后一项。我们通常需要操作系统（例如 Ubuntu Trusty 14.04）、编程语言（例如 Python 2.7）、监控代理（例如 New Relic System Monitor 5.1.93）、配置代理（例如 Chef 客户端 12.0.0）、过程管理程序（例如 Monit 5.10）、Web 服务器软件（例如 Apache 2.4.10）、版本控制软件（例如 Git 2.0.1）、安全软件（例如 Snort 2.9.7.0）、日志记录软件（例如 logstash 1.4.2）、SSL 认证、密码和 SSH 密钥——只有一切都准备好之后，才开始安装应用程序代码。

我们还必须确保每一种软件都正确无误，包括正确的版本，否则应用程序也许会出故障。你的公司可能有数以百计，甚至数以千计的服务器，取决于上面运行的软件类型（例如，应用程序、数据库、队列、负载均衡服务器），每台服务器的配置也许是不一样的。手工安装和维护这一切很耗费时间且容易出错，使用定制的 shell 脚本会好一点，但仍然会很杂乱，所以我们最好的选择就是使用文档齐全、久经考验且最好是开源的**配置管理系统**。

配置管理可用来描述几种类型的系统，包括应用程序配置、虚拟机、容器和编排工具。

1. 应用程序配置

大部分应用程序会提供一些调整选项，让我们无须修改应用程序代码就可以对应用程序进行调整，比如日志设置、内存设置和端口数量。通常可以指定不同环境中使用的不同配置（例如开发环境、准备环境、生产环境）。举例来说，在 Ruby on Rails 中，应用到所有环境中的设置是放在 config/application.rb 文件中的：

```
config.i18n.default_locale = :en
config.assets.compress = false
```

若要修改生产环境中的某些设置，可以把覆盖的设置放到 config/environments/production.rb 中：

```
config.assets.compress = true
```

尽管你可以在一个地方进行设置的调整、不需要触及代码，甚至能够在应用程序之间共享设置（这些都很重要），但这样可能会太灵活。例如，LinkedIn 服务提供的不同的配置参数是在 10 000 这样的数量级，每个参数在不同的环境中都有不同的值。配置的巨大数量和频繁的交互导致了配置 bug 常态化地出现。

> 根据我们的经验，修改配置信息没有修改源代码那么危险，可以说一直都是一个虚构的神话。
>
> ——Jez Humble 和 David Farley，《持续交付》

我们要用对待代码的方法对待配置数据：把它存储在文件中，置于版本控制之下，对所有修改进行评审和测试。

2. 虚拟机

虚拟机（virtual machine，VM）镜像就像是一个正在运行的操作系统的快照，上面已经安装了所需的各种软件。我们可以在**管理程序**内运行 VM 镜像，这样的管理程序有 VMWare、VirtualBox 和 Parallels，它们将底层的硬件抽象出来。这样的话，不管所在的服务器是什么，运行在 VM 镜像内的软件看到的都是完全相同的环境。这意味着我们可以在开发环境中定义一个 VM 镜像，上面具有所有软件，镜像在生产环境中也以完全相同的方式运行，这降低了“在我的机器上没问题”这样的 bug 出现的概率。不幸的是，VM 镜像是重量级的，会导致启动时间的开销以及 CPU 和内存的消耗。

3. 容器

容器就像一台 VM，我们可以在其中定义容器镜像，安装好需要的所有软件。这个镜像在所有环境中都以完全相同的方式运行，包括部署环境和生产环境。然而，和 VM 不同的是，容器是非常轻量级的，它直接运行在现有的操作系统之上，但能够让自己的进程、网络连接和文件系统完全与底层的环境隔离。因此，容器镜像启动的速度非常快，并可实现最小的 CPU 和内存开销。

最流行的容器工具是 Docker，它可以让你在不到一秒钟内启动一个隔离的 Linux 镜像。Docker 可以轻松在开发和生产环境中运行完全相同的镜像。而且由于镜像非常轻量级，我们可以在同一台机器上运行多个镜像——比如一个用于数据，另一个用于应用服务器。其缺点则是容器技术和特定的操作系统有密切的关系。举例来说，Docker 只能运行在 Linux 上，因为进程、文件系统和网络隔离功能都是基于 LxC（LinuX Containers）和 Linux cgroups[8] 而实现的。

4. 编排工具

编排工具是一些通用的自动化工具，可以用来定义如何配置服务器。流行的编排工具包括 Chef、Puppet、Salt 和 Ansible。要使用这些工具，我们需要定义所有要管理的服务器，通过手动方式或者通过一种服务发现机制，再编写一些脚本或方法步骤，定义每台服务器应该如何配置。例如，使用 Ansible 的话，可以在 /etc/ansible/hosts 文件中定义服务器：

```
[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

然后再定义若干服务器**角色**。这是一些 YAML 文件，列出配置服务器执行该角色需要的任务。例如，下面是一个 `webserver.yml` 角色，指定了一台 Web 服务器所需要的配置：

```
- name: Install httpd and php
  yum: name={{ item }} state=present
  with_items:
```

注 8：在开发过程中，你可以使用虚拟机，实现在 OS X 和 Windows 上运行 Docker。现在甚至也有轻量级的 Linux 发布版专门用于运行 Docker，名为 boot2docker，它可以在几秒钟内启动，仅使用 25MB 的内存。

```
    - httpd
    - php

- name: start httpd
  service: name=httpd state=started enabled=yes

- name: Copy the code from repository
  git: repo={{ repository }} dest=/var/www/html/
```

最后，创建一个可执行的 `playbook`，让 Ansible 把 webserver 角色应用到 /etc/ansible/hosts 中标记为 `webservers` 的所有主机：

```
- hosts: webservers
  roles:
  - common
  - webserver
```

像 Ansible 这样流行、开源的编排工具提供了大量辅助库，可以用一行代码去处理许多常见的配置管理任务，比如安装软件、复制代码和提供特定环境的配置。这样就可以**像对待代码一样**对待基础结构，即通过人可读的文本文件去定义服务器，这些文件可以存储在版本控制系统中，可以对它们进行代码评审、共享和测试。

5. 部署自动化

简单定义什么软件应该放在服务器上是不够的，我们也需要定义如何把它放上去。例如，升级单台应用服务器一般包括以下几个步骤：通知监控系统（例如 Nagios）服务即将下线、从负载均衡（例如 HAProxy）的轮循中将服务移除、把新的代码安装并发布到服务器上、重新把它加回到负载均衡轮循中、通知监控系统服务又上线运行了。我们需要在每一台应用服务器上都运行这些升级步骤，且最有可能以滚动的方式进行。例如，假设总共有 20 台服务器，可能会一次升级 5 台，让另外的 15 台继续运行，并提供实时服务。

在很多不同类型的服务器上部署很多不同类型的代码可能会变得很复杂。所以，这些工作永远不要手动去做，部署过程和构建过程是一样的，应该是自动的，才能够用单独一条命令去运行它。这不仅对实现可扩展的开发过程是必不可少的，对于实现可重复、可测试和高质量的开发过程也同样不可或缺。

我们可以编写定制的 shell 脚本去管理自动化任务，但是最好还是使用开源的、久经考验且有丰富文档的库去代替。最后的选择就是上面介绍的编排工具，包括 Chef、Puppet、Salt 和 Ansible。例如，上一节介绍了如何在 Ansible 中用 playbook 创建和应用角色，下面再介绍如何扩展这个 playbook，让它实现滚动升级：

```
- hosts: webservers
  user: root
  serial: 5

  pre_tasks:
  - name: disable nagios alerts for this host webserver service
    nagios:
      action: disable_alerts
      host: {{ inventory_hostname }}
      services: webserver
    delegate_to: "{{ item }}"
```

```
  with_items: groups.monitoring

- name: disable the server in haproxy
  shell: disableServer.sh myapplb/{{ inventory_hostname }}
  delegate_to: "{{ item }}"
  with_items: groups.lbservers

roles:
- common
- webserver

post_tasks:
- name: Wait for webserver to come up
  wait_for:
    host: {{ inventory_hostname }}
    port: 80
    state: started
    timeout: 80

- name: Enable the server in haproxy
  shell: enableServer.sh myapplb/{{ inventory_hostname }}
  delegate_to: "{{ item }}"
  with_items: groups.lbservers

- name: re-enable nagios alerts
  nagios:
    action: enable_alerts
    host: {{ inventory_hostname }}
    services: webserver
  delegate_to: "{{ item }}"
  with_items: groups.monitoring
```

其中 `serial: 5` 命令让 Ansible 进行滚动升级，同时把这些修改应用到 5 台服务器上。`pre_tasks` 指定了应用任何角色前要运行的任务，可以用它们去禁用 Nagios 和 HAProxy。`post_tasks` 则指定了之后要运行的任务，可以用它们重新启用 Nagios 和 HAProxy。同样，这种方法可以让我们像对待代码一样对待基础结构。

8.4.3 持续交付

现在我们弄清楚了部署在哪里（主机）、部署什么（配置管理）以及如何部署（部署自动化），最后的问题就是何时部署？答案是：什么时候都可以。

持续交付是一种软件开发规程，以此构建可以随时发布到生产环境中的软件。我们可以每天发布，也可以一天发布几次，甚至每次通过自动化测试签入之后发布（即所谓的**持续部署**）。持续集成让开发人员编写可以频繁合并的代码，相互之间保持同步；持续交付让开发人员编写可以频繁部署的代码，保持和生产环境的同步。

> 程序员桌面上的东西和生产环境中的东西有任何不同，就都是风险。在与已部署软件不同步的情况下，程序员面临着做出决定而又无法获得这些决定的精确反馈的风险。
>
> ——Kent Beck、Cynthia Andres，《解析极限编程》

如果你使用的是容易出错、手动的、部署缓慢的开发过程，“一天要进行好几次”听上去就很痛苦。但是和本章介绍的其他内容一样，如果它伤害了你，就意味着你需要更加经常地做这件事。然而，为了让持续交付安全且切实可行，我们必须先做到支持**回滚**和**向后兼容性**。

1. 回滚

即便我们进行了许多自动化测试并将新的功能放到功能开关中，还是会有 bug 漏过。处理生产环境中的 bug 的一种方法就是**回滚**，即部署版本较旧的代码。与回滚相对的就是前滚，即尝试快速发布带修正补丁的新版代码，而这是有风险的。回滚会花几分钟的时间，但修复一个 bug 可能要花长得多的时间，而且无法保证新的代码就能够真的修复那个 bug 而不引入新的 bug。对于持续交付而言，回滚通常是更好的选择，因为可以立即修复问题，如果找到了修复的方法，部署也不会花太长时间。

我们还可以遵循**金丝雀部署**模型，实现更安全的部署和回滚。当你部署新版代码时，先部署在单台服务器上，这就叫金丝雀。其他所有的服务器继续运行老的代码，之后可以把金丝雀服务器和老的服务器（作为基准）进行对比，查看是否存在 bug 或性能问题。例如，LinkedIn 有一个叫 EKG 的工具，可以比较金丝雀服务器和基准服务器，并自动突出显示它们在 CPU 用量、内存用量、延迟、错误和其他指标上的差异，如图 8-5 所示。

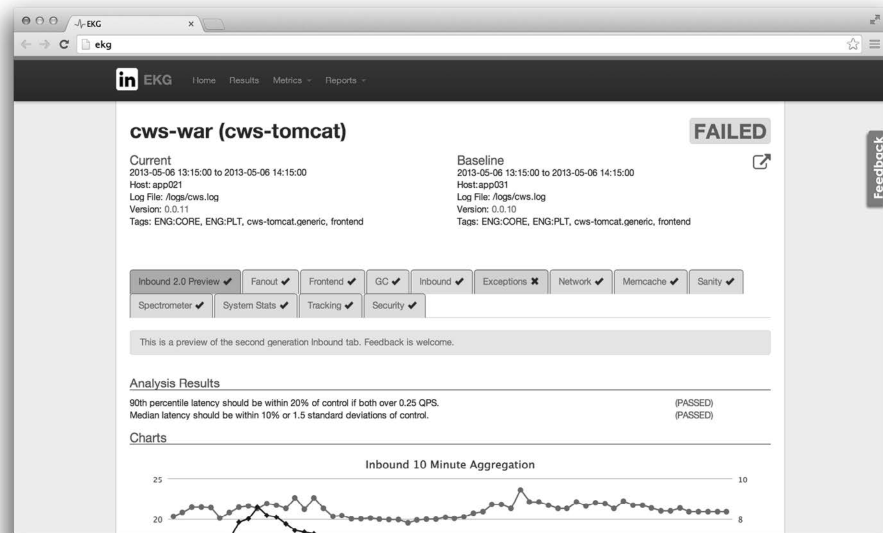


图 8-5：使用 EKG 比较金丝雀服务器和基准服务器

如果金丝雀服务器存在任何问题，它们也只会影响一小部分用户，我们可以对单台服务器进行回滚去修复问题。如果一切看起来都没问题，在几分钟之后，就可以把新版代码部署到其他所有服务器上。

2. 向后兼容性

向后兼容性似乎永远都是分布式系统的需求，但它对于持续交付特别重要，因为服务也

许会在任何时候以任何顺序部署或回滚。实现向后兼容性有两条通用规则。

(1) 作为服务，我们不能在没有功能开关的情况下**删除**公开 API 中的任何东西。
(2) 作为客户端，我们不能在没有功能开关的情况下**依赖**公开 API 中的任何新东西。

举例来说，如果我们开发一个服务，它的公开 API 是 RESTful，并返回 JSON，规则 1 意味着添加一些新的东西（比如新的 URL 或新的查询字符串参数）通常是安全的，但不能删除或修改 URL，也不能重命名任何查询字符串参数、重命名 JSON 中的任何字段，或者修改这些字段中的任何类型。从公开 API 中去掉某物的唯一方法就是把移除操作放在一个功能开关中，只允许使用数为零之后才开启。对于客户端来说，规则 2 意味着另一个服务中任何对新 API 的调用必须放在一个功能开关中，且只能在新的 API 完成部署之后开启。

8.5 监控

> 如果你无法测量，你就无法修复。
>
> ——David Henke，LinkedIn 技术、运营高级副总裁

实现代码并部署到生产环境中还不够，我们也要确认代码在部署之后能够持续工作，这就是引入监控的原因。这里的监控指的是所有用于看到代码和用户实际情况的工具和技术，比如日志文件、Google Analytics 和 Nagios。可以这么说，监控为阴，单元测试为阳。

> 测试通常就是对少数情况下正确性的强假设进行检验，而监控则是对实际生产负载下正确性的弱假设进行检验。
>
> ——Jay Kreps，Confluent 联合创始人

和单元测试一样，在代码中添加监控不仅可以帮助我们找到 bug，也可以发现一些设计上的缺陷。难以监控的代码通常也是难以阅读、维护和测试的代码。因此，在我们实现的基础结构中，对代码的测量应该和编写代码测试一样简单而寻常。不幸的是，设置监控可能是很困难的，因为监控领域可以说是各自为政。

在编写本书的时候，市场上有数以百计的开源和商业监控产品，但是没有一种能提供全面的解决方案。占据优势地位的是 Nagios，它创建于 1999 年，而且具有配套的 UI。2011 年前后，Twitter 上出现了 #monitoringsucks 的主题标签，而且很快就被加入到 GitHub 的工具、指标、博客文章和诉苦等分类的资源中，帮助人们宣泄不满，希望能够解决监控的问题。这并不是为了打击 Nagios，因为它在很多方面也做得非常好，而只是为它敲响警钟——建立全面的监控解决方案仍然是一个超乎人们想象的复杂任务。

本节将讨论监控的不同方面，包括日志记录、指标和报警。

8.5.1 日志记录

```
// 真正的程序员如何调试
println("*************************** here")
```

如果服务器上出现了问题，日志文件通常是第一个知道的。日志记录是最基本、最普遍也最好理解的监控形式。就像所有编程语言都有记录日志的库，比如 Java 的 log4j。尽管

上面是个笑话，但我们总要使用真正的日志记录库，而不是 println。我们要掌握的关于日志记录最重要的事情就是日志名称、级别、格式和聚合。

1. 日志名称

大部分日志记录框架都提供了为日志记录器命名和单独配置每个日志记录器的灵活性。例如，使用 log4j 时，可以调用 LogManager.get 去指定日志记录器的名称：

```
public class CheckoutPage {
  private final Logger logger = LogManager.get(this.getClass());

  public void loggingExample() {
    logger.info("Hello World");
  }
}
```

注意，这里使用了一个类作为名称（this.getClass）。这是一种应当遵循的良好实践，完整保留下来的类名将出现在每一条日志消息中，我们可以轻松找出该消息是从哪里来的：

```
2012-11-02 14:34:02,781 INFO [com.mycompany.CheckoutPage] - Hello World
```

可以为每一个命名的日志记录器提供自定义配置，比如将日志输入写到不同的文件中：

```
log4j.appender.com.mycompany.CheckoutPage.file.File=/logs/checkout.log
log4j.appender.com.mycompany.HomePage.file.File=/logs/home.log
```

有了上面的配置，当我们在 checkout 页面调式问题时，可以看到 checkout.log 中的日志消息，不会因主页的日志消息而分心。

2. 日志级别

大部分日志记录库都支持不同的日志级别，比如下面是 log4j 的分类：FATAL、ERROR、WARN、INFO、DEBUG、TRACE，按重要性从高到低排列。比如，FATAL 用于需要立即关注的严重问题：

```
logger.fatal("The program crashed!");
```

而序列的另一端则是 TRACE，通常只用于开发期间低级的诊断。比如，下面是实现 println("******** here") 消息更好的方法：

```
logger.trace("Entering method foo");
```

我们可以将应用程序中每一个日志记录器设置为不同的日志级别：

```
log4j.logger.com.mycompany.CheckoutPage=ERROR
log4j.logger.com.mycompany.HomePage=TRACE
```

在上面的配置中，我们将 CheckoutPage 日志记录器设置为 ERROR，意味着只有该级别或该级别以上的消息（即 ERROR 和 FATAL）才会显示在它的日志中，所有其他级别的消息都会被忽略。日志文件可能会变大，特别是对处理每秒数以百计请求的服务器而言（可以解释为每秒有数千条日志条目）。将日志条目限制为 ERROR 及以上级别可以确保在正常操作期间，日志文件只会包含最可能识别问题的信息（例如非预期异常的栈跟踪信息），降低由于消息被不重要的日志条目淹没而导致错过关键问题的概率。一旦找出了具体的问题，就可以针对相关子系统使用更加宽松的日志级别（例如在上面我把 HomePage 的日志记录器设置为 TRACE），从而在调试时提供更多日志记录的信息。

3. 日志格式化

日志文件有两大受众：开发人员和工具，这意味着日志文件的格式必须是可供人阅读和对工具（比如 grep）友好的。大多数日志记录系统都可以让我们指定应用到每一条日志消息的**模式**，所以不需要手动设置。举个例子，下面是 log4j 的模式：

```
%d %p [%c] - %m%n
```

这一模式告诉日志记录器，每一条消息都要包含日期、日志级别、线程名、记录器名、破折号、日志消息和一个新行。所以下面的日志消息：

```
logger.info("Hello");
```

在日志中将显示如下：

```
2012-11-02 14:34:02,781 INFO [com.mycompany.CheckoutPage] - Hello
```

以下是日志消息格式化的一些最佳实践。

包含时间戳

对于找出事情发生的时间（5 秒前还是 5 天前）和事情发生的顺序（对于并发性的 bug 调试特别有用）来说，时间戳是必不可少的。我们应该在时间戳中包含尽可能多的时间粒度，从年一直到毫微秒。

包含唯一的 id

有了唯一的 id（比如 GUID），我们就有可能将多条有关联的消息联系在一起。例如在 Web 服务器中，每个请求一般都有一个唯一的 id。如果这个 id 存在于每一条日志消息中，我们就可以很容易地对日志进行过滤，找出在处理某一请求时发生的所有事情。

让文本对开发人员友好

使用简单、简洁的话语，避免使用缩写，为每条消息提供足够的上下文，让人们在没有看到代码的时候也能理解，例如下面这条日志消息：

```
2012-11-02 14:34:02,781 - 12345; 200; chkt;
```

再对比这条：

```
2012-11-02 14:34:02,781 - userid=12345; status=200; url=/checkout
```

你更愿意在调试的时候看到哪条？

让文本对 grep 友好

大多数开发人员使用像 grep 这样的工具在日志文件中搜索，这意味着几乎所有日志消息都应该单独成行并利用一些容易解析的分隔符，比如 `key=value` 对。

记录全栈跟踪日志

“单行”规则唯一的例外就是栈跟踪。如果代码的某个部分抛出异常，哪怕会跨越多行，也要把整个栈的跟踪信息放到日志中，因为它包含了一些必备的关键调试信息，能够弄清楚问题从何而来。

4. 日志聚合

把日志消息写入文件中既简单又有用，但也存在两个限制。第一个限制就是大小：如果

我们不断写入同一个文件中，文件最终会变得太大而没什么用，甚至可能耗尽磁盘空间，让服务器崩溃。大部分日志记录系统都能够配置**日志轮循**，使日志记录器可以在现有文件超过某一大小时创建新的文件（例如，checkout.log 一旦超过了 10MB，将被重命名为 checkout-page-09-01-2014.log，一个新的 checkout.log 将会生成），也可以在日志文件已经存在超过一个可配置的时间段之后将其删除（例如删除超过两周的日志文件）。第二个限制是可访问性的限制：如果你有一台服务器，通过 SSH 去读取日志文件是不成问题的；但是当你有十几台或上百台服务器时，理解上面的所有日志可能是非常困难的。

为了解决可访问性的问题，现在出现了许多工具，比如 syslog、logstash 和 flume。这些工具可以把所有服务器的日志聚合起来，以一种有用的方式去组织信息，比如将其加载到 Hadoop 或搜索索引中。现在也出现了一些将日志管理作为服务提供给用户的公司，比如 Splunk、Sumo Logic、logstash 和 Papertrail。例如，loggly 可以自动把所有服务器上的日志聚合起来，提供一个 Web 界面让我们对所有日志文件进行快速搜索，并生成日志统计信息和图形的仪表板。

8.5.2 指标

实现日志记录之后，监控的下一个步骤就是收集指标。可以收集的指标有很多不同的类型，每种类型都对应不同的工具和服务，我将根据详细程度对它们进行划分，分别是：可用性、业务、应用程序、进程、代码和服务器。

1. 可用性

可用性是每一家公司都应该测量的最基本的指标。用户是否能够访问你的产品？这是一个是或否的问题。也许 Web 服务器下线了，也许负载均衡器不能工作，也许移动应用中存在一个 bug，也许网站由于数据库超负荷而变得太慢。但从用户的角度来说，他们并不关心这些，他们只知道要么一切正常，要么不正常。

要监控产品的可用性，必须有现实中的测试用例，这些用例要运行在客户使用的界面上，根据产品的不同，可能是网站、移动应用或 API 端点。你可以设置自己的服务对可用性进行监控，但是让产品倒下的问题（例如负载均衡器的问题或者数据中心断电）同样也可能让你的可用性监控出问题。Keynote 和 Pingdom 这样的第三方服务也许会是更好的选择，因为它们是专门监控可用性的，可以从世界各地许许多多不同的位置去监控正常运行时间。

2. 业务

业务指标是对用户所做事情的度量，比如页面浏览、广告展现、销售、安装，或者其他一些对业务很重要的指标。这些是 CEO 和产品团队所关注的指标，如果数值突然下降，我们必须要尽可能快地知道。现在也有许多工具可以跟踪业务指标，比如 Google Analytics、KissMetrics、MixPanel 和 Hummingbird。

3. 应用程序

业务指标之下是你的应用程序代码。对于客户端应用程序而言（比如网站或移动应用），要使用真实用户监控（Real User Monitoring，RUM）工具（比如 Google Analytics、Keynote、New Relic 和 boomerang）去跟踪负载大小、加载时间、错误和崩溃等情况。对服务器应用程序而言（比如 Web 服务器、数据库、缓存、队列和负载均衡器），可以使

用像 New Relic 和 AppDynamics 这样的工具去跟踪 QPS、响应时间、吞吐量、请求和响应大小、URL 命中、响应代码和错误计数等指标。这一级正是进行日志记录的地方。

4. 进程

每个应用程序都由一个或多个需要运行的进程构成。不幸的是，现实中会出现进程崩溃和服务器重启的问题，所以我们需要进行一些额外的工作，确保能够重启那些进程。现在有许多**进程管理程序**，可用于监控进程并对其进行重启，这样的工具有 Monit、God、Upstart、supervisord、runit 和 bluepill。

5. 代码

应用程序之下就是我们编写的代码。有许多有用的指标可以用来跟踪代码库，比如代码的行数、bug 的数量、构建次数和测试覆盖。该领域比较实用的工具包括我们自己的构建系统、CI 服务器（例如 Jenkins 和 Travis）以及代码分析服务（比如 Code Climate 和 Codacy）。

6. 服务器

最后进入硬件层。在这一级，我们要测量 CPU 开销、内存开销、硬盘开销和网络流量这样的指标。该领域的主流工具是 Nagios、Icigna、Munin、Ganglia、collectd、Cacti 和 Sensu。

8.5.3　报警

日志和指标只在有人关注的时候才是有用的。鉴于时间短、日志数据多、指标多种多样，大多数监控数据都是没人去看的。因此，为了让监控真正发挥作用，就需要设置报警——当出现了需要关注的东西（比如服务器不可用）时，能够自动收到通知。上面提到的许多监控工具都可以让我们定义一些规则，设定通知应该在什么时候发送出去。例如，可以定义“如果应用服务器 QPS 与上周相比下降超过 20%，则向邮件列表中的地址发送通知”或者“如果服务停止响应，则发送短信给待命技术人员”。PagerDuty 和 VictorOps 这样的服务可以帮助我们管理待命轮循、通知（例如通过 email、IM、短信或电话）以及将通知逐级上报的过程。

报警的关键点是要能够识别出问题产生的原因。一些指标和日志有时候拥有我们需要的所有信息，但如果不具备这样的信息，当问题突然出现时首先要问的就是“什么发生了变化？”要回答这个问题，一个**变更管理面板**会比较有帮助，它可以展示对产品的所有修改。8.4 节提到的许多配置管理和部署自动化工具都提供了 Web 界面，可以展示所有近期的部署和修改情况。另一种选择就是把所有工具（包括部署、bug 跟踪、A/B 测试和源代码控制）连接到一个中心服务上（比如 Slack 或者 HipChat）。

8.6　小结

> 敏捷需要安全。
>
> ——Jay Kreps，Confluent 联合创始人

Facebook 的口号曾经是“快速行动，打破陈规”[9]。结果表明这是很容易实现的，而“缓

注 9：Facebook 的新口号是“快速行动，稳定架构”。

慢行动，墨守成规”也是如此。真正有风险的是“快速行动，墨守成规”。我们可以犯错，但过于频繁地打破陈规最终将拖慢脚步，因为我们必须花更多时间处理旧的问题，而不是实现新的东西。换句话说，让我们能够快速变化的其实是我们知道哪些变化是安全的。

第 6 章和第 7 章介绍了如何编写整洁的代码，并安全地进行修改。但是在生产环境下，代码编写之后的过程也同样重要，包括要通过版本控制和构建过程将代码和其他开发人员的代码集成起来；通过建立主机、配置管理和自动化部署，让代码进入生产环境中；通过日志记录、指标和报警等方式，保持代码在生产环境中的持续运行。所有这些任务以前都是由单独的运维团队处理的，但是在过去的这几年，DevOps 运动的出现鼓励开发人员和运维人员之间进行更有效协作。尽管 DevOps 现在有点滥用，但让开发人员更多地参与到构建、部署和监控之中仍然有非常实际的好处。

在 LinkedIn，持续集成、持续交付和全面监控的实施使得技术部门可以更快地行动，即便在不得不应对上千万新用户和数以千计的新员工的情况下也能行动自如。我们从痛苦的、容易出错、两周一次的发布过程走出来，进入一小时内就可以多次发布出 bug 更少的代码的境界。结果表明，要快速前进，就需要确信修改不会有多少损害。这正是我们在自动化测试、功能开关、金丝雀发布、回滚和其他 DevOps 实践中有所投入之后才能收获的。

DevOps 运动的主要好处是可以让开发人员意识到软件在“代码完成”或“QA 认证”之后仍未完成。软件永远没有完成的时候。现如今，软件就是活生生的、会呼吸的东西，会持续生长和进化，所以我们需要像持续集成、持续交付和全面监控这样的过程，时常去检查核实它仍然活着并运行良好。

> 如果软件不是像折纸飞机——折好之后就立刻放飞，那样被“做出来”的呢？相反，如果我们把软件更多地看作一种有价值，能生产，需要养育、修剪、收割、施肥和浇灌的植物呢？传统的农夫知道如何让植物在数十年甚至数百年都能持续生产。如果我们也以同样的方式对待程序，软件开发会有何不同呢？
>
> ——Steve Freeman、Nat Pryce，《测试驱动的面向对象软件开发》

第三部分

团队

第 9 章

创业文化

9.1 要行动，不要口号

> 文化并不是桌式足球桌或信任背摔。它不是政策，也不是圣诞舞会或公司野餐。这些都只是物体和活动，不是文化。文化也不是沙龙。文化是行动，而非口号。
>
> ——Jason Fried、David Heinemeier Hansson，《重来》

由员工分享并通过他们的举止和行动表达出来的信念、设想和原则，构成了创业文化。为什么文化很重要？因为文化胜过战略。创业要想成功，仅有出色的点子、绝妙的规划，甚至伟大的产品都是不够的，因为即便是最出色的点子、规划和产品，最后都免不了失败。相反，我们要做的是一家伟大的公司，而每一家伟大公司的核心都是伟大的文化—— 一种能够让我们想出新点子、新规划和新产品的环境。

一些开发人员会回避文化这样的“软”话题，但是他们有时又会抱怨自己的工作没有得到认可、觉得自己做的项目无意义、办公室太吵、苛刻的老板不断因为一些无意义的会议打断他们导致工作无法完成。文化并不是软话题，它是公司这一含义中非常核心的内容。记住，创业是与人密不可分的，公司中没有什么东西比文化对人的影响更大。

本章将讨论如何塑造创业公司的文化。首先，我会解释如何定义核心理念，这是由公司的使命和核心价值构成的。我还将描述如何把这种理念落实到公司的方方面面，包括组织设计、招聘与晋升、激励、办公室、远程办公、沟通和过程。对于每一个话题，我都会用许多有启发性的实际例子去介绍伟大的文化应该是什么样的。

9.2 核心理念

> 我从商业中学到的最有价值的一课就是，管理一个超速增长的公司就像发射火箭——如果发射时失之毫厘，进入轨道就会差之千里。
>
> ——Jeff Weiner，Linkedin CEO

如果你的创业公司是火箭，使命就是火箭的目的地，它告诉人们公司存在的意义。对于任何目的地，你都会想到有许多不同的轨道，这就是核心价值的源泉，它们是用来决定**如何**完成使命的原则。把使命与核心价值放在一起，就形成了公司的核心理念——处理事情的指导原则。

作为创始人，我们也许想参与公司的每一个决定。但随着公司的成长，我们要参与决定的数量是以指数级减少的。如果员工都根据不同的原则自己做决定，就好比尝试让火箭同时向不同的方向发射——它不可能走得太远。因此，我们应该花些时间提前定义指导原则（核心理念），使每个人都可以用这些原则去做决定，让火箭朝着同一个方向发射。

定义公司核心理念的一种方法是把团队的领导层关在房间里几天，让他们充分探讨，做出决定。另一种选择就是询问所有员工——这是 Zappos 使用的方法，CEO Tony Hsieh 每一年都会向全公司发送电子邮件征求意见。

> 我们想把 Zappos 的文化做成一本小册子，作为新员工的入职培训资料之一。我们的文化是把所有员工对公司文化的理解结合起来，所以我们会把每个人的想法都放在这本小册子中。
>
> 请把你对 Zappos 文化的理解用 100~500 个词描述出来并发邮件到我的邮箱。（什么是 Zappos 文化？它和其他公司的文化有何不同？你觉得我们的文化怎么样？）
>
> ——Tony Hsieh，《回头客战略》

所有的回复，通常包括许多故事和照片，都会被收集到“Zappos 文化手册”并发给每一位员工[1]。通过让员工亲自写下并参与整个过程，公司的使命和价值得到了加强和固化。

我们来更深入地探讨一下使命和核心价值。

9.2.1　使命

使命宣言清楚地表达了公司的目的。它应该解释公司为什么存在，公司是做什么的、为谁而做。我们应当把使命当作公司的指南——它是一个梦想、一个目标，它是你、你的员工和客户所向往的。

下面是几个比较好的例子。

> Google 的使命是整合全球范围内的信息，使人人皆可访问并从中受益。
> Facebook 的使命是给人以分享之动力，让世界更为开放、更为紧密相连。
> LinkdeIn 的使命是连接全球职场人士，助他们事半功倍、发挥所长。

下面是几个比较差的例子。

> 沃尔沃：通过为客户创造价值，从而为股东创造价值。我们利用专业能力，创造出质量上乘、安全、环保的交通工具和服务。我们以活力和热情对待工作。我们尊重每一个人。

注 1：Zappos 也把它们的文化手册放到了网上。

> Twitter：通过信息共享和传播平台产品，让每个人与世界紧密相连，从而触及世界上最大的日常受众，成为世界上收入最高的互联网公司之一[2]。

这些好坏之间有什么区别呢？好的使命宣言应该简洁、清晰、永恒并鼓舞人心。

1. 简洁

使命宣言不应该超过一两句话。它应该重点突出、简单并且好记。许多人都能轻松凭借记忆背出 Google 的使命，但是你能记住沃尔沃的使命宣言吗？或者 Twiiter 的呢？它的宣言甚至都放不进一条推特中。

2. 清晰

从公司的使命宣言中，我们应该能够理解公司存在的原因以及它想要做的事情，不需要多问就能明白。更重要的是，使命宣言应该是公司的独特标识。LinkedIn 和 Facebook 都是社交网络，但是从 LinkedIn 的使命宣言中可以清晰地了解到该公司和专业人士、职业、工作有关，我们永远不会把它和 Facebook 的使命宣言相混淆，而 Facebook 的宣言则是关于分享和联系。作为对比，沃尔沃的使命宣言满是陈词滥调，可以用在任何公司，尽是为股东创造价值、以活力和热情对待工作这样的话。

3. 永恒

不要把你的使命宣言和当前的战略或产品混淆起来。战略和产品是**如何做**和**做什么**的问题，而使命宣言则是**为什么**的问题。“如何做”和“做什么”可以根据不断变化的世界相应地改变，但是“为什么”在整个公司的生命中将是始终如一的。举个例子，我们注意到 Google 的使命宣言并没有提到搜索。搜索是他们为了“整合全球范围内的信息，使人人皆可访问并从中受益”而采取的当前战略，但是在未来，这一战略也许会是完全不同的，可能会涉及手机、可穿戴设备或者无人驾驶汽车。但不管他们在做什么产品，他们“整合全球范围内的信息，使人人皆可访问并从中受益”的使命是不会变的。

4. 鼓舞人心

公司的使命必须是某种能让你充满激情的东西，某种可以让你有目标感，并成为让你每天醒来和不倦追求的原因。如果该使命不让人感到足够重要，客户是不会去购买的，投资者也不会投钱，更无法让你的团队维持创业所需的专注、活力和坚韧。这是因为使命宣言回答的是所有问题中最重要的一个：为什么。

在 TED 的演讲“How Great Leaders Inspire Action”中，Simon Sinek 介绍了“黄金圆”的概念，如图 9-1 所示。

Sinek 解释了大多数人做事情都是由外而内的。举个例子，在尝试销售产品的时候，你也许会先从你“做什么”开始：“我们做了一个 Y 功能的产品 X。”接下来也许会谈到你“怎么做”：“我们的竞争优势是 Z。”但是很少有人知道他们“为什么要做”。这里的为什么不是指“为了赚钱”，而是某种更加伟大的目标。大多数人不知道这一目标是什么，所以他们在讨论为什么这一问题的时候都非常模糊，甚至根本就不去讨论。

注 2：Twiiter 在 2014 年的分析师日上发布了这一宣言，引发一阵吐槽，后来很快就解释这实际上是“战略宣言”，而非“使命宣言”。不管是不是使命宣言，都可以作为有用的例子。

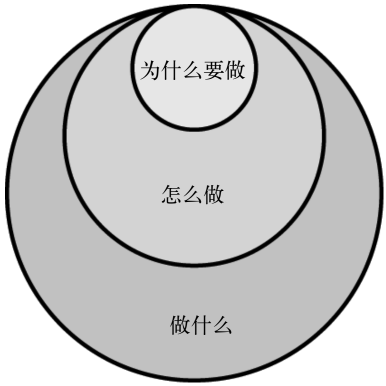


图 9-1：黄金圆

作为对比，伟大的公司和伟大的领导者则是从相反的方向开始做起的。他们先从“为什么”开始，并强烈地关注这一问题。“怎么做”和“做什么”仅仅只是支持的细节——它们是用户完成“为什么”的特定方法，后者才是至关重要的。Sinek 用 Apple 给出了一个很好的例子。

> 如果 Apple 和别的公司一样，他们的推广信息听起来可能是：“我们制作出色的电脑。它们设计精美、使用简单、界面友好。想买一台吗？”
>
> “无聊。”
>
> ……
>
> Apple 实际上是这样表达的：“无论做什么，我们都坚信在挑战现状。我们信仰用不同的方式思考。而我们挑战现状的方式就是开发出拥有设计精美、使用简单、界面友好的产品。我们于是制造出了最棒的电脑。你想要买一台吗？”
>
> 完全不一样，对吗？你已经准备从我这里购买一台电脑了。我所做的只是将这些信息的顺序重新排列。这些证明了人们不想从你那里买你所做的产品，人们想买的是你的信念和宗旨。
>
> ——Simon Sinek，How Great Leaders Inspire Action

从“为什么”开始，从你的使命开始，这是作为领导人的最强大之处。这样可以更容易地获取客户，有助于招聘，也有助于筹集资金。请记住，马丁·路德·金做的演讲是“我有一个梦想”，而非“我有一个计划”。当你的产品正在艰难的孕育当中，你的团队一周要工作 80 个小时，这就是梦想，这就是说服你们保持前进的理由，而肯定不是一个计划，或是获得收益的预期去驱使你们。所以为什么说公司的使命不应该和金钱有关。

客户不会愿意在使命陈述中看到关于股东价值、收益和利润这样的东西，因为这些只会让人们觉得公司是贪婪且不值得信赖的。同样也不会有员工愿意为了提高 2.3% 的利润率而将心血投入到项目中。当然，每家公司都是要赚钱的，但金钱并不是公司的目标，它只是让公司达成真正目标的资源。金钱就像氧气，它是维持生活所必不可少的，但并不是生活的目的。事实上，忽略金钱是想出一个好的使命陈述的最佳方式。

> 如果你明天醒过来，银行里就有多到可以再也不用工作的钱，我们要如何设定组织的目标，才能够让你可以继续工作下去？什么样的更强烈的目标感才能够激励你继续将宝贵的创造力投入到公司的成就当中？
>
> ——Jim Collins、Jerry I. Porras，《基业长青》

这里隐含着一个出人意料的道理：有时，一个大胆、宏大的使命实现起来并不会比一个小一点的使命容易。任何创业都是艰难的，但如果这一事情的回报是完成一个宏大而重要的目标，可能更容易说服出色的人参与进来。为出色的人提供一个宏伟的挑战才能够让他们在职业中尽其所能。

有史以来最为大胆的使命宣言就出自约翰·F. 肯尼迪总统：“我相信国家将会齐聚一心完成这一目标——在十年之内让人登陆月球并安全返回地球。”假如肯尼迪是一名CEO，也许会这样说：“我们的使命是以团队为中心，在最大程度上进行创新，将航天主动权作为战略目标，成为太空产业的国际领导者。”幸好，他在 1961 年的国会演讲上选择的是前面那条宣言。仅仅八年之后，尼尔·阿姆斯特朗和阿波罗 2 号的宇航员就出现在了月球上。

9.2.2 核心价值

> 至关重要的一点是，员工要理解公司不仅关心结果，还关心结果是如何获得的。
>
> ——Ed Harness，宝洁公司前主席

核心价值是你在组织中用来做出每一个决定的信条。核心价值并不需要去选择，因为它们在你的团队中即可发现。它们是你已赖以为生的主要价值，你对其深信不疑，它们对你至关重要，不管你加入什么公司或者做什么产品，都会坚持这些价值。

不要把核心价值和文化规范、管理风潮或者当前策略这些弄混。它是一个核心价值，是哪怕你不再工作也有足够的钱却仍会在生活中遵循的价值，是即便一百年以后，即便你加入完全不同的公司，即便你因遵循这一价值而被市场惩罚，也会遵循的价值。举例来说，假设你在硅谷工作，考虑把“宽松的着装规定”作为公司的核心价值。结果会怎样呢？几年之后，你跑到华尔街去一决高下，那里所有的人可都是西装革履。你还会坚持“宽松的着装规定”这一政策吗？如果不会的话，它就仅是你遵循的文化规范，不应该成为你的核心价值。另一方面，你是否笃信“透明沟通”？如果在华尔街竞争，你的公司还将保持透明吗？在那里保密就是规范，所以透明化也许只是核心价值的一个不错备选。

理想情况下，你可以让所有人在房间里，想出你们所有人共享的核心价值是什么。如果公司人数已经太多，你可以尝试组建一个“火星小分队”。做法如下：假设要发送一支5~7 人的团队到火星上创建新的子公司，你会挑选谁？当你考虑这样的任务时，自然而然会挑选出对公司的各个部分有深入理解的那些人来，给他们机会让他们清晰地表达对公司核心价值的看法。

一旦找到了你们的核心价值，就把它写下来，发给所有员工。甚至还可以公开去分享你们的价值标准，这样可以帮助你找到遵循类似价值的员工、客户和投资者。例如，Netflix以其文化简报而闻名，截至 2014 年，它在 Slideshare 上已经有将近一千万次的浏览量。

当然，定义你的使命和核心价值仅仅只是迈出第一步。下一步，或者可以说接下来的

一百步，就是有意地规划你的公司，把它的核心理念具化到每一次行动中。接下来的几节将讨论让公司围绕其使命和价值开展行动的一些最重要的方法，包括组织设计、招聘和晋升、激励、办公室、远程办公、沟通和过程。

9.3 组织设计

> 组织设计的第一法则就是所有的组织设计都是不好的。无论什么设计，都是以其他部分为代价来优化组织某些部分之间的沟通的。例如，如果你把产品管理放到技术部门中，优化了产品管理和技术之间的沟通，代价是弱化了产品管理和市场推广部门之间的沟通。因此，只要你提出新的组织设计，人们就会找出它的缺点，觉得自己的想法才是正确的。
>
> ——Ben Horowitz，《创业维艰》

我们必须对公司的组织结构进行设计，使每一名员工都清楚地知道自己的职责所在，高效地开展工作，而他们的工作汇聚起来又能推动公司向着使命前行。当公司还很小的时候，这些自然都是很明确的，所以我们可以忽略组织设计。创始团队只需要设定方向，每一名员工跟着前进即可。随着员工数量的增长，这种方法的效率也会降低。

效率上的缺失是由于复杂的项目无法被划分为完全可以独立完成的单独任务，所以在沟通和协调上总是存在开销。在不存在任何组织结构的情况下，对于一个有 n 个人的公司，每个员工必须和 $n-1$ 个员工协调。举例来说，一个公司有 3 个人，每个员工就必须和另外 2 个人协调；但是如果是 100 人的公司，每个员工就必须和另外的 99 人协调。因此，总的沟通开销将以 n^2 增长，用不了多长时间就会对生产效率带来显著的影响。实际当中，在公司超过 20~30 名员工的时候，就需要进行一定的组织设计以保持顺畅运转。

从较高的层次看，可以实施的组织设计有两种类型：一种是传统的**经理驱动等级结构**，另一种则是尝试打造**分布式组织**。

9.3.1 经理驱动等级结构

大部分人对经理驱动法都比较熟悉，因为它是多数现代公司所使用的。其理念是将公司分解为若干小团队，每个团队都向一位经理报告并关注一个单独的任务（例如分为技术团队、销售团队和产品团队）。团队中的每个成员通常只需要和所在团队的其他成员协调，不需要和整个公司协调。当多个团队需要一起工作的时候，大部分协调都在经理们之间进行。由于每一名经理的直接下属的数量被控制在一定范围内，公司是按照经理逐层上报直至 CEO 的等级结构设计的。

在分层级的组织中，经理负责大部分的协调和决策。当然，经理的作用远不止于此。彼得 · 德鲁克被认为是现代管理的奠基人，他写下了一名经理应当开展的五项基本工作。

设定目标

经理应当决定一些目标、为了实现这些目标需要完成什么工作，以及如何将这些目标传递给团队。

组织

经理应当对工作进行划分，并挑选人员从事这些工作。

激励和沟通

　通过在薪水、人员安排和晋升方面的决定以及经常性的沟通，经理应当将人员凝聚成一个高效的团队。

评估

　经理应当为每个人建立目标并评估他们迈向这些目标的进度，形成一种绩效评估方式。

助人成长

　经理应当帮助团队中的每一个人提升自己的能力。

经理驱动等级结构型组织的管理方式有许多优点。不同层次的管理层使谁有权利和责任进行决定变得很清晰。员工知道他们要向谁报告，谁来评估他们的绩效。团队在特定的任务上会变得高度专业和高效。最后，这是一种可以被很好理解的、久经考验的方法，已经被证明在大规模的公司中是有效的，比如像沃尔玛这样拥有 220 万名员工的公司。

层级组织也有一些缺点，部门之间的沟通容易变得困难，组织的多层级化也可能导致许多官僚主义方面的开销，从而降低了效率。经理也可能成为所有沟通和协调的瓶颈所在。报酬、名声和赞赏通常都和你所在的层级紧密关联，便也导致许多人痴狂地关注如何往上升，醉心于弄权而非做事。最终，高级别的经理，也就是被赖以信任进行最重要决定的人，通常都远离了实际工作，在最不合适的地方去做决定。

9.3.2　分布式组织

层级设计的另一种形式就是分布式组织（有时也称为**扁平组织**）。其理念是让员工针对每个任务自我组织，形成最有效率的结构，在任务完成或改变之后再重新组织。根据任务的不同，也许会有不同的人去承担管理层和协调角色，但都是根据任务的需求而定的，没有固定的头衔或者严格的层级。最出名的例子就是一家名为 Valve 的视频游戏公司。

> 层级有利于维持可预测性和可重复性。它简化了计划的制订，更容易自上而下控制一大帮人，这也是军事组织极其依赖这种方式的原因。但如果你是一家娱乐公司，在过去 10 年竭尽所能去招募地球上最聪明、最有创造力和最有天赋的人，却让他们坐在办公桌前并告知他们要做的东西，那就湮灭了他们 99% 的价值。我们要的是创新者，这意味着我们要维护一个任由他们挥洒的环境。
>
> 这就是 Valve 采取扁平化的原因。我们用这种方式直截了当地表达了我们没有任何管理，每个人都不需要向其他任何一个人报告。我们有创始人或主席，但他也不是你的经理。这个公司是由你们操控的——驶向机会，远离风险。你有权利为项目开绿灯，有权利发布产品。
>
> ——Valve

如果你终身都在大公司工作，必然对管理层级习以为常，这样的方式看起来可能是混乱且行不通的。你真的能依赖人们去自我组织吗？大多数人在没有经理背后盯着的时候不会偷懒吗？其实这两种顾虑都是基于工作场所中人类行为的假设而得来的，但这些假设在至少过去 50 年很大程度上都是错的。

Douglas McGregor 在 1960 年《企业的人性面》一书中，讨论了一种普遍的观念，即一般的工人并不喜欢他们的工作，并极可能去避开它。对某些工作来说也许是这样，比如一

些无聊的、重复的手工劳动。那种情况可能需要一直有管理人员的存在，用报酬（薪水、奖金）促使工人们工作，或者用惩罚（解雇、蒙羞）威胁他们。然而，这种方式在许多需要创造性的工作中并不适用，比如编程。许多人喜欢使用自己的创造力去制作东西，所以他们实际上是喜欢自己的工作的，而且会积极主动地寻找要解决的问题。而我们要做的就是为他们提供合适的环境，他们内在的驱动力将促使他们努力工作。

换句话说，管理并不是商业运行的基本需要，而是一种发明。管理，正如我们今天所知道的，是在 20 世纪早期发展起来的，是为了迎合那个年代商业的需要。它只是一种工具，和所有的工具一样，需要不时更新以满足新的需求。今天使用的大部分管理实践都是设计用于组织从事手动劳动和组装线上的工人，让他们做重复的、无聊的任务的。其主要的目标就是为了"把人变成半程序控制的机器人"。而现代高科技创业公司的需求则有很大的不同，所以无法保证同样的管理实践也能发挥作用。

在 Valve，员工们自行组织成为"小团队"，即各种多专业交叉的项目团队，而不是依赖于经理。如果人们认为一个项目的重要性值得参与，他们就会加入团队——而不是因为老板让他们必须加入。所有的办公桌都有轮子，他们可以按照自己挑选参与的项目定期移动办公桌，形成新的团队。这种方法使 Valve 获得了不可思议的成功，他们做出了史上卖得最好的四个 PC 游戏，2014 年被评为工作最让人满意的游戏公司，2011 年的估值达到 20 亿 ~40 亿美元，意味着如果按每个员工所赚得的钱来算，这家 400 人的公司已经超过了 Google 或 Apple。

其他公司也在尝试分布式组织。2013 年，Zappos 宣布他们采用了一种名为 Holocracy 的实践，这种实践去除了许多传统的层级结构、经理和头衔。Holocracy 也被其他几家公司采用，包括 Medium 和 David Allen 公司。

> 研究表明，每当城市的规模扩大一倍时，每名居民的创造力和生产力就会增加 15%。但是当公司变得更大时，每名员工的创造力和生产力一般都是下降的，所以我们要尝试找到办法，希望像城市一样去设计 Zappos 的结构，而不像官僚的公司。在城市中，人和商业都是自组织的。我们尝试从正常的层级结构切换到一个叫 Holacracy 的系统上，从而实现和城市管理一样的效果，允许员工们表现得更像创业者，能够自己决定工作的方向，而不是向一位告诉他们要做什么的经理报告。
>
> ——Tony Hsieh，Zappos CEO

分布式组织的另一个例子是开源软件。像 Linux、Apache 和 MySQL 这样的项目表明，让世界各地数以千计的开发人员在没有集中办公的情况下，一起工作并生产出复杂而又成功的项目是可能实现的。这些项目也证明了分布式项目的质量和开发速度可以高于采用传统经理驱动等级结构的商业公司所能达到的水平。9.7 节将讨论更多关于开源的实践。

扁平和分布式组织的优点是员工有更多的自主权和责任心，后文将会介绍，这些都是激励的基本要素。更多的自主权意味着最接近问题的人，也就是最了解问题的人，能够做出重要的决定，避免层级结构中大部分官僚主义带来的恶开销。不幸的是，分布式组织在公司领域的运用相对还比较少，所以不像管理驱动等级结构那样好理解和阐述。这就意味着我们需要找出替代方法，应对 Drucker 的五个管理任务。

设定目标

在许多分布式团队之间协调大型的修改和新的优先级顺序可能会很棘手。

组织

在分布式组织中，对于每一个决定应该谁来负责或有权执行并不总是很明显。层级式管理已经可以扩大到数以百万计的员工，但是却几乎没有例子表明分布式组织是否也能达到这一规模（尽管对大多数创业公司来说这一点也许无关紧要）。

激励和沟通

没有激励，我们就需要用不同的体系去评估员工，比如同级审查。没有层级，就需要有晋升的替代方案，9.4 节将讨论这个问题。

评估

每个团队现在都负责确定自己的目标和里程碑并跟踪进展情况。

助人成长

在不存在固定的团队或经理的情况下，要建立起人员招聘和培训的有效过程可能是很困难的。

那么，我们应该在创业时选择层级结构，还是扁平组织呢？其实没有正确的答案。和大多数有趣的问题一样，组织结构设计也是一个权衡取舍的游戏。如图 9-2 所示，几乎每一种类型的组织结构设计都有以此为基础而建立的成功的公司，每一种类型也都有自己的优缺点。

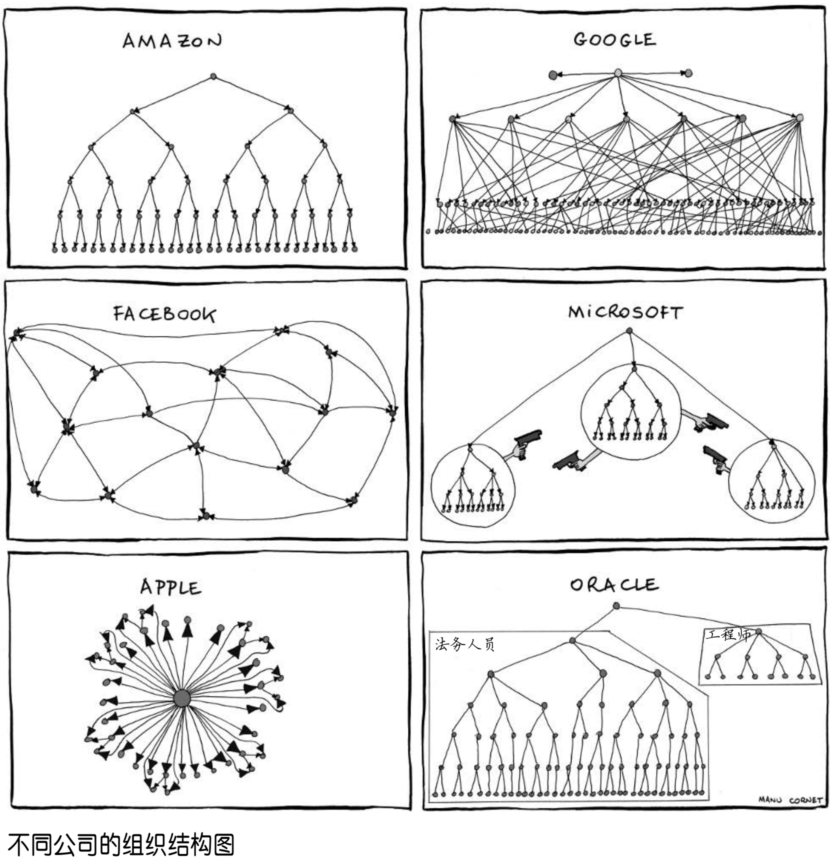


图 9-2：不同公司的组织结构图

9.4 招聘与晋升

让组织和使命、价值保持一致的最佳方法就是招聘已经与之相匹配的人。这就是为什么说招聘过程必不可少的一点就是，找到有很好的**文化契合**的人。第 11 章将讨论文化契合以及创业公司招聘的其他各个方面的内容。

在这一章，我唯一想提出的就是恳求所有科技创业公司都要把保持多样性作为你们的文化和招聘策略的一个核心。女性和少数族裔在当今科技行业的从业人数远远不足。举例来说，女性在所有 IT 相关的专业工作中只占 25%，而其中的 56% 又会在其职业生涯中半途离开科技行业，退出率是男性的两倍。作为创业公司，应该有意地寻求在技能、性别、种族和背景上的多样性，不仅仅因为这是一件正确的事（确实是），也因为这么做将给你带来竞争上的优势。研究表明，多样性能使团队更愿意做出新的尝试，更有创造性，更可能去分享知识并帮助公司触及更多的客户，吸引更多的人才，做出更加创新的产品。数字是不会撒谎的：执行董事会人员多样化的公司在收入上也明显更高，女性董事占比最高的公司比女性董事最少的公司在股本回报率上要高出 53%，在销售回报率上高出 42%，在投资资本回报率上高出 66%。

除了要做到文化契合之外，招聘还要不断地对那些成为公司核心价值典范的员工进行奖赏，鼓励这种与公司价值一致的行为。在大多数公司中，传统的奖赏方式就是晋升，让员工获得新的头衔、新的职责和提升。这样做也会导致职业阶梯的出现，员工要通过组织得到提升，一次晋升一级，直到顶端。职业阶梯有两点需要引起我们注意：彼得原理和以管理作为晋升。

9.4.1 彼得原理

彼得原理阐述了在层级组织中，每个员工都趋向于被提升到他们不能胜任的级别上，最终“每一个职位都将被一个不能胜任其工作的员工所占据”。其推理过程很简单：如果你在当前的角色中表现出色，最终会被提升到新的角色。终有一天，你也许会被提升到超过你能力的角色，无法表现得足以继续晋升，便困在这个无法胜任的角色中。

如果我们使用晋升作为奖赏，彼得原理就是必然的结果。为了避免这一结果的出现，我们需要做些额外的工作，比如只晋升那些已经在新角色上履职的员工，并实施培训计划帮助他们做得更好（阅读第 12 章了解更多信息）。然而，技术组织中彼得原则发生最常见的原因是将开发人员晋升为管理者。

9.4.2 以管理作为晋升

许多软件公司相信，奖励开发人员的最佳方式就是将他们提升到管理层。问题是，管理岗通常并不是由技术岗逐级上升达到的，它是需要完全不同技能的一份工作。精通算法、架构和测试并不意味着你在安排工作的优先次序、进行复杂的人际关系谈判和培养人才方面也会做得很出色。在大多数情况下，你失去的是一位非凡的贡献者，但得到的却是一位平庸的经理，而且他十分可能会让其他个体贡献者的前进步伐也慢下来。

以管理作为晋升的最坏影响是向你的技术组织传递了这样一个信号：编写代码是二等工作。想把一家科技公司弄垮，最快的方法就是建立起这样的组织，只让没有经验和表现

不好的开发人员去编写代码，而让所有有才能的熟练开发人员反过来做管理者的工作。我们无法通过管理获得好的代码，好的代码需要出色的开发人员来编写。

为了保证那些出色的开发人员能够继续编写代码，我们需要找到一种奖赏技术领导能力的方法——通过代码贡献在公司中赢得影响力，而不是依赖对人的管理。实现的一种方法就是为个人贡献者（individual contributors，IC）和管理人员提供同等的职业阶梯。举个例子，表 9-1 展示了 LindedIn 的职业轨迹。

表9-1：LinkedIn职业阶梯

个人贡献者	管理层
工程师	经理
高级工程师	高级经理
首席工程师	主管
杰出工程师	高级主管
研究员	副主席

理论上，不论你进入的是哪一条轨道，每一个等级所获得的报酬、承担的责任以及在公司中的影响力应该是大致相同的。实际当中，由于体系的偏见，这是难以实现的。有一种偏见是管理层头衔在大多数社交活动中声望更高。成为“主管”或者“副主席”是了不起的事情，成为“高级首席工程师”就没有那么厉害了。另一种偏见源于经理通常拥有个体贡献者（即便是同一级别的）并不拥有的权力，比如了解公司的路线图、人员的薪水，并控制着招聘、解聘等过程。这将形成一种权力鸿沟。最终，当现有的管理人员被太多的直接下属弄得不堪重负时，机构会分层，管理人员通常会得到晋升，这种情况在成长中的公司是定期发生的。另一方面，个体贡献者通常只会因为出色的工作受到认可而被提升，出现的频率就没有那么高了。

结果就是，在大多数科技公司中，高级管理人员（主管及以上级别）的数量要比高级技术人员（首席工程师及以上级别）多得多。在 LindedIn，高级管理人员的数量是高级技术人员的 5 倍左右。这个数字在 Etsy 也是类似的。

> 我是 Etsy 第一个拥有“首席工程师”头衔的人，这是一个和管理职位相对应的技术职位（即比 CTO 低一级）……一度，我是这一级别唯一的一人，但在这期间却有五个在理论上和我级别一样的主管，这一比例至少在较低的职位上是不合适的（我不知道这家公司现在还是不是这种情况，也许已经不是这样了）。
>
> 要想讲得通，我们就必须相信一些经不起推敲的事情。首先，我们必须相信技术人员存在需要被管理的高度倾向，我觉得这和我们的预期相违背。我们中没有哪几个是怀着不用实际做事情的希望而进入公司的。
>
> 其次，我们必须相信尽管需要五位主管才能有效地管理组织，但只要有一位技术领导就可以对同一组人每天所做的工作详细情况给出建议。
>
> ——Dan Mckinley，Etsy、Stripe 软件工程师

这样的一种不平等传递出来的信息是，即便个体贡献者拥有专门的通道，管理仍然是一种更好的职业建议。这个问题也没有简单的解决方法，如果你打算在公司中使用职业阶梯，为个体贡献者设置单独的通道仍然比什么都不做要好，但你需要密切关注管理人员

和 IC 的比例。你甚至应该在个体贡献者攀爬职业阶梯的过程中赋予他们更多的权力，尝试去消除这一权力鸿沟，比如对产品的控制权、对基础结构的决定权以及选择自己项目的自由权。

另一种替代的选择就是完全不使用职业阶梯。在大部分扁平、分布式组织中，晋升并不是一种奖赏。对于表现出色的人仍然需要奖赏，但并不是关注怎么去给他一个花哨的新头衔或者让他更上一层楼，关注的是如何赋予他新的责任，给他新的报酬和好处。有点像用奖状和勋章奖赏军人，而不是晋升他们的军衔。例如，我们可以让高级工程师更能影响公司的发展方向，更能控制他们想要从事的项目，给予他们更多的时间去发展自己的技能，以及更多的薪水、股权和假期。

公司可以永远不使用职位头衔、职业阶梯和晋升吗？这很难说。在《创业维艰》一书中，Ben Horowitz 认为大多数创业公司最终将会出于两个原因而引入职位头衔。第一，因为其他大部分公司都在使用这些头衔，你的员工也需要一个头衔，以便他最后可以换工作。第二，职位头衔是“公司中角色描述的速写”，员工、客户和合作伙伴经常需要用这些头衔和公司打交道，特别是随着公司规模的扩大。尽管在大多数公司中，头衔都是一种标配，但少数公司甚至直接就把它取消了：CloudFlare 的 100 名员工都没有头衔，Valve 的 400 名员工也没有头衔，Zappos 正尝试取消 4000 名员工的头衔。

9.5 激励

大多数公司使用晋升和奖金去激励员工。在《驱动力》一书中，Daniel Pink 提出的研究表明，只有在任务无聊且重复的情况下，奖励完成任务的员工才能提高他们的表现，比如一些手工劳动。如果任务需要创造性，比如编程，提供奖赏实际上可能会影响员工的表现。这一与我们直观感受不同的结果有一个最有名的例子，来自于一个与蜡烛有关的实验。呈现在实验参与者面前的是一根蜡烛、一盒大头钉和一盒火柴，如图 9-3 所示。实验的目标是找到将蜡烛固定在墙上的方法，可以点燃它并且不让任何蜡滴到下方的桌子上。在接着往下读之前，读者不妨花一分钟想想你要怎么解决这个问题。

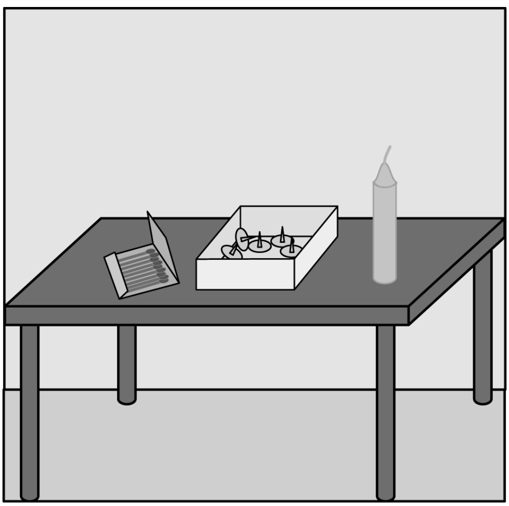


图 9-3：蜡烛问题

你是否会尝试直接用大头钉将蜡烛钉在墙上？也许你还会想融化一些蜡用作固定物？不幸的是，这两种解决方法都不可行。正确的解决方法如图 9-4 所示，就是把大头钉的盒子清空，用大头钉将盒子固定到墙上，再把蜡烛放到盒子里。

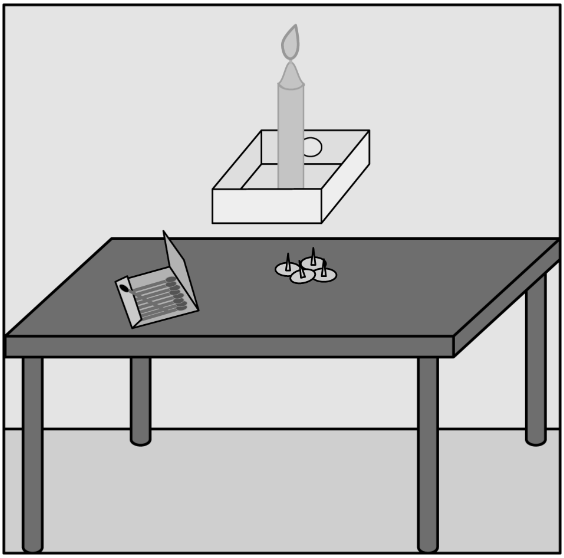


图 9-4：蜡烛问题解决方法

当这个问题被用在实验中时，参与者会被分成两组。A 组被告知为了收集这个练习的统计数据，会给他们计时；而 B 组则被告知最快想出解决方法的人将会得到金钱奖励，所以也要计时。哪一组的表现更好呢？传统的管理和经济理论都预测 B 组的表现更好，但结果证明，B 组实际上比 A 组平均多花了 3~5 分钟去解决这个难题。提供金钱的激励反而会影响表现。这是因为解决蜡烛问题需要创造性，你必须克服**功能固着**，意识到这个盒子不仅是用来存放大头钉的，而且是可以解决难题的道具。

为创造性的任务提供奖励也许会影响表现，其中的原因就是，你会将注意力放到奖励而不是任务上。以每年典型的绩效评估过程为例，假如你是经理，你和你的直接下属 Anna 坐在一起，告诉她："Anna，如果你在 11 月前发布这个新的移动应用，你就会加薪！"这会导致若干使 Anna 消极的心理学副作用。

- 你给 Anna 传递的信号是，快速实现这个移动应用肯定是痛苦且让人不悦的——否则你为什么必须为此提供奖励呢？
- Anna 要成功完成此任务将会感到许多压力，一方面是因为她知道她的经理想完成任务，另一方面则是因为她想得到奖励。这种额外的压力也许会让她感到紧张，并降低了她的创造性和表现。
- 你给 Anna 的指示是她的首要目标是加薪，而不是实现一个高质量的移动应用。对奖励的渴望和对成功的压力也许会诱使她不惜代价按时发布应用，哪怕走走捷径或钻钻空子。
- 如果 Anna 无法按时发布这个移动应用而导致没有加薪，也许就会扼杀她的积极性。如果那一年她努力工作却什么都没得到的话，她会感到被背叛了。

- 即便 Anna 按时完成了这个应用，你也给她加了薪，现在你可以确信的是如果不继续允诺加薪，Anna 永远不想再做另一个移动应用了。事实上，下一次加薪必须比前一次加薪的幅度更大才会成为真正的激励。奖励就像毒品一样会使人上瘾。这也是为什么许多人会因为别人为他们的爱好支付了报酬，就失去了对爱好的热情。

简而言之，大量研究表明，公司为了提升员工表现而最常使用的方法——为任务完成而承诺给予加薪、晋升或奖金这样的奖励，通常都会产生相反的效果。当然，不是所有奖励都是不好的，在你考虑激励和自我实现这样的高层次需求之前，需要先考虑马洛斯的需求层次理论的前几个层次，比如房租和食物。因此，你需要提供基本的薪水。此外，大多数人实际上并不清楚自己的激励究竟是什么，所以他们会寻求更多的金钱，即便金钱实际上并不能带来快乐。因此，在最开始招聘一个人的时候，我们仍然必须提供有竞争力的薪酬方案（阅读 11.5 节了解更多信息）。然而研究表明，除了基本的报酬之外（每年大约 75 000 美元），更多的钱，或者更多其他的**外在激励因素**并不必然会带来更多的激励。

那么要怎么办呢？答案就是**内在激励因素**——我们都具有的内部驱动力，能够让我们因为任务本身的利益而喜欢上一个任务。人天生就是爱玩而好奇的，我们经常会在一些困难的任务上投入大量的时间，即便没有人为此买单。如果你是程序员，你已经有所体验了。你是否曾经花了一个周末的时间去改动一个工作之外的项目？你是否曾在工作之余学习过一门新的编程语言或技术？你是否曾为解开朋友告诉你的一个逻辑谜题而无法入眠？你是否曾经向开源项目做过贡献？这些活动大部分都是受内在激励因素驱动的。

如何才能促进内在激励呢？由于它是很自然的，我们无法从外部去驱动内在激励。我们能做的只是提供一种环境，有利于将人们已经具备的内在激励带动出来。为此，我们需要营造一种能最大程度放大自主权、专业能力和目标的环境。

9.5.1 自主权

自主权是我们为控制自己生活而与生俱来的驱动力。我们想要界定自己做什么工作、何时工作、如何工作以及和谁一起工作。如果你想鼓励员工发挥自主权，就需要招聘可以信任的人，为他们设定清晰的目标，然后放手让他们去干。这就是为什么很多创业公司关注的是结果而不是如何去得到它。你可以根据自己的意愿决定工作几个小时，可以去办公室或在家工作，可以以自己想要的方式去实现项目，只要你能实现分配给你的目标即可。如果你可以挑选自己的目标，自主权的驱动力甚至会变得更大。

有一种低成本、低风险的方法可以对这种想法进行试验，就是在一些特定的日子让员工可以做他们想做的任何事情。LinkedIn 每个月举行一次“黑客日”（hackdays）、Facebook 每几个月举行一次“黑客马拉松”（hackathons）、Atlassion 每个季度举办一次“派送日”（ShipIt Day），这种形式在各个地方大致都是相同的：你先想出一个点子，组成一个团队，在 24 小时内做出这个点子的“解法”或者快速原型。这是一种很好玩的活动，提供了许多食物、咖啡，并有机会向整个公司展示你做的东西，公司也会对最好的解法进行奖励。2010 年，Facebook 还引入了“黑客月”，只要在公司超过一年的员工都可以加入他们所选择的另一个团队，和他们工作一个月。

> 我参加过第一次“黑客月”。这真是一次很好的活动，可以给公司带来许多好处。它使团队具备容错的能力。在任何时刻，任何人都可以离开公司、离开团队，他们也可以消失，怎样都行。所以“黑客月”是一种检验团队是否太过依赖某个人的低风险方法。它带来的另一个好处就是跨文化交换，公司中的每个团队都会擅长做不同的事情，有的团队也许善于进行代码评审、有的团队也许善于测试、也有的团队也许精于 UI 和 UX。所以希望在你切换团队的时候，也可以把这些优势随之带走。你也可能是派到其他团队的使者：“嘿，你们的测试做得太糟糕了，我们正打算自己来做单元测试。”当然，这种方式还能够促进团队之间的人员流动，对于优秀的人才来说，他们可以更好地在公司内部更换团队，而不用完全离开公司。
>
> ——Daniel Kim，Facebook、Instagram 软件工程师

少数公司甚至会为员工提供更多的自主权。19 世纪 50 年代，3M 公司倡导“15% 时间”政策，允许技术人员花 15% 的时间从事他们所选择的项目。到现在，3M 的业务支柱中有很多发明最初都是在“15% 时间”的项目中研发出来的，包括便利贴、胶带等。Google 也提供了一项类似的政策，名为“20% 时间”。员工已经因其创造出了许多成功的产品，包括 Gmail、Google News 和 Google 翻译。Valve，就是我前面提到的那家视频游戏公司，把这种想法甚至又推进了一步。

> 我们已经听说过其他公司让员工分配一定比例的时间到自主项目中。在 Valve，这一比例是 100%。
>
> 因为 Valve 是扁平化的。人们不会因为被告知加入什么项目而加入项目。相反，你可以决定先问自己几个合适的问题（后面再介绍），之后再决定要做什么。员工会用脚（或桌子的轮）为项目投票：令人信服的项目就是大家都可以看到其价值的项目，这些项目很容易招募到员工。这意味着我们时常会进行一些内部招聘活动。
>
> ——《Valve 新员工手册》

9.5.2 专业能力

专业能力是把事情做得更好的内在驱动力。大部分程序员都在找机会学习新的技术或者磨炼他们的技能。我们天生就会被困难的问题吸引，因为我们知道它将给我们带来挑战，促使我们做得更好。我们读书、浏览博客和文章去学习最佳的实践。在很大程度上，编程就是一门手艺，每一名程序员都想成为大师级的匠人。

每个公司都想招聘到最出色的程序员，但是最好的公司都会紧紧抓住他们的程序员并在他们身上投入，让他们变得更加出色，这就形成了一种良性循环。这些员工会因为他们正在提升自己的技能和市场价值而高兴，公司也因为得到更出色、技术更精湛的员工而高兴。专业能力上的投入有许多方法，比如鼓励员工定期加入新的项目和团队，参加会议和演讲（能够在会议和演讲中有所表现则更好），组织公开论文、博客和图书阅读小组（能够自己写论文、博客和图书则更好），向开源项目做贡献（阅读第 12 章了解更多信息）。

另一种在专业能力上投入的方法，来自 Reid Hoffman、Ben Casnocha 和 Chris Yeh 所著的《联盟》一书。在多数公司，面试的过程、录用通知书以及经理和员工交谈的方式都是基于一种共同的错觉，认为该员工一直都会在公司工作。实际上，在一家公司工作 50 年，不断沿着职业阶梯攀爬，带着金表退休的日子早已不复存在了。《联盟》一书背后的思想是雇主和员工都应更早地面对这一现实。我们应该约定一个**任期**，而不是把一份工作的录用看作是一辈子的合同。所谓任期，就是在有限长度内明确定义的使命，员工承诺在这期间通过对具体项目的贡献投身于公司，而公司则承诺通过帮助员工提升他们的市场价值，从而实现在员工身上的投入。举例来说，也许可以约定“在 12 个月内发布新的移动应用”这样一个任期；作为交换，公司将培养你的 iOS 和移动设计技能，从而提升你的市场价值。当这一任期结束，你可以重复和公司的对话，约定一个新的任期。

通过这种方法，我们得以用更加透明的方式代替模糊、不确定的员工评估过程。而且它也不仅仅提供外在的奖励（例如薪水），还提供了内在的奖励（比如公司投资你的专业技能），这是一种明显的交互。LinkedIn 的全局方案高级副总裁 Mike Gamson 写过一篇博客，提到了一个很好的例子，这篇博客名为“My Promise to You (Our Employees)”

> 我们先从坏消息讲起。坏消息就是有一天你会离开 LinkedIn。我知道你才进入公司就要考虑离开是很奇怪的，但我要让你关注这件事，这样我们才能最充分地利用我们一起在公司的时间，携手前行。我不清楚你打算在这里度过 2 年、5 年、10 年还是更长的时间，但我要确保无论你和我们一起在这段旅途上待多长时间，当你在 20、30、40 年后回首整个职业生涯的时候，你会觉得待在这儿的岁月是职业生涯中变化最大的。你在这里的这些年学到的最多、成长最快、接触到最多不可思议的人和最具创新性的想法。我希望你在这里的这些年能成为真正改变你职业轨迹的岁月。我希望你在 Linkedln 比在你们本可能选择的其他地方，获得生命中更多的成就。我向你承诺，我将致力于营造和培育能够让你改变职业轨迹的环境，为你提供接触各种思想、各种人员、各种体验的机会，满足你在生命中实现这一切所需的因素。当你在某天离开 LinkedIn，我希望你能够真正地转变。我所期望得到的回报是你能允诺全身心投入到这个自我转变的机会中，投入到我们的公司和这个世界中，并用你的勇气和坚持去追寻这些机会。
>
> ——Mike Gamson，LinkedIn 全局解决方案副总裁

9.5.3 目标

> 如果你想做一艘船，就不要召集人们去收集木材，也不要给他们分工或下命令。相反，你要让他们对广阔无尽的大海产生渴望。
>
> ——Antoine de Saint Exupéry

目标是我们从事一些有较大意义的事情的内在驱动力。我们都渴望能够实现一些超越自身的事情。我们想让自己在这个世界上留下印迹。我们需要赚钱使这一目标成为可能，但是金钱并不是真正的关键——它只是帮助我们实现其他一些目标的资源。

在本章开头，我谈论了确定一个有感染力的公司使命的重要性。我探讨过人受**“为什么”**而驱动的事实，表达的意思其实就是人是受目标驱动的。你的使命越能够鼓舞你，你就

越能更好地传递这一使命，也就越容易激励你的员工。举个例子，回想一下在讨论绩效评估时，你让 Anna 在 12 个月内发布一个移动应用以获得奖励。假设你改成这么说："Anna，这个移动应用是你能触及这个星球上拥有智能手机的 20 亿人的生活的机会，也是你学习 iOS 和移动开发技能的机会，更是你让整个公司朝着它的使命再进一步的机会。"

给 Anna 一种目标感可能比给她奖金这样的奖励更能激励她。那么，是否仍然有办法可以给她奖金而又不会扼杀她的积极性呢？是的。关键就是要避免使用**有条件奖励**——根据"如果－那么"这样的条件去奖励，比如"如果你做了 X，我就给你 Y"。你把任何奖励挂在人们面前，就是吊杆上的萝卜，自然就成了一种外部激励。员工会以奖励为**交换**去做工作，这样你就会看到前面讨论的失去动力的各种副作用。然而，如果奖励是意料之外的，这些副作用中的大部分就会消失[3]。和允诺给予奖励不同，你要营造一种环境，让 Anna 关注任务本身并受内在的激励因素驱动（例如公司的使命和她对学习 iOS 以及移动开发技能的渴望）。如果她做得很好，你可以给她一个意想不到的**认可**，而不是给她一种预期的**报酬**。

此外，我们应该开展持续的轻度评估，可以每周进行一对一的会谈，而不是每年进行一次绩效考核。通过这种方式，我们可以立即对出色的表现进行奖励，而不是等到一年之后。这种奖励不一定总得是金钱。事实上，最好的奖励通常是更多的自主权、更多的专业能力和目标。比如可以给 Anna 更多自由去选择她的下一个项目，让她转到其他团队并学习新的技术，赞扬她并给她公开的认可，让她知道自己的工作是很重要的，经常表达你的谢意，庆祝重要的里程碑，专门留出预算用于团队定期旅游和出行，哪怕只是旁边的一家比萨店。

当然，加薪、奖金和股份的提升也是很重要的，只要它们不属于有条件的奖励，而只是表示感谢的一种好方法。发放像奖金这样的外在奖励有一个好方法，就是把它们和内在奖励结合起来。例如，我们可以在整个公司的员工面前弄个仪式发放奖金，而不是在一对一的会面中私下发放。这样的仪式可以体现出人们做重要事情的内在驱动力，并且赢得同事的认同。这也是许多竞赛所起的作用：真正的奖品并不仅仅是金钱，而是其他人看到你是胜者。

9.6　办公室

感受公司文化最好的方式之一就是走入它的办公室。例如，我们来看看 GitHub 在旧金山的总部。当你第一次走进去的时候，迎面而来的是几个玻璃展柜。一个里面有一尊青铜雕像，是史前动物章鱼猫的骨骼（章鱼猫是 GitHub 的 logo）。另一个里面有一台笔记本电脑，是其中一位创始人用来实现第一个 pull 请求的。当你走过展柜，会进入到 GitHub 的等候室。但它并不只是等候室——它复制了总统办公室（见图 9-5）。那里有一张很大的木桌、一面美国国旗、一块带有 GitHub 标识的巨大圆形地毯（一只拿着橄榄枝和餐具的章鱼猫）、绒毛躺椅、排列着旧书的书架。

注 3：行为研究表明，最有效的奖励行为是不定时的，即随机对一些努力和尝试进行奖励，但不是全部。想想拉斯维加斯的老虎机，你知道你肯定会赢，但不知道是什么时候，所以才会长时间地玩下去。



图 9-5：GitHub 旧金山办公室的等候室

穿过等候室，就是一个巨大的开放区域，包括完整的酒吧、自助餐厅、足球桌、乒乓球桌、台球桌和 DJ 电台。这就是 GitHub 的员工们吃午餐、举行聚会和放松的地方。而真正的工作则是在上一层进行的。在那里，你会看到开放式的布局，许许多多员工在那里开发。有些人坐着，有些则使用立式桌子，少数人的脚边还有狗在休息。在角落有一个图书室，塞满了各种技术图书。在它的旁边有一间会议室，但它又不是会议室，而是战况室。在房间的中央，摆着一张八角木桌，周围是一圈皮椅，还有大电视、美国国旗和几个世界时钟（见图 9-6）。



图 9-6：GitHub 战况室

青铜雕塑、酒吧、总统办公室看起来也许很奢侈，但不妨这样想：如果你是全职工作，每年在办公室要待上 2000 小时以上，几乎是清醒时间的一半。所以，很难说花那么多时间在自己喜欢的办公环境中可以值多少钱。营造一个出色的工作场所本身就是值得的，因为它能取悦现有的员工，也可以吸引新的员工。而且对大量人群的研究清楚地表明，好的办公室设计可以显著提升生产效率。

开发人员的理想办公室需要满足 4 个条件：

(1) 一个可以和他人一起工作的地方；
(2) 一个可以独处专注工作的地方；
(3) 一个可以放下工作的地方；
(4) 一种可以根据个人需要定制办公室的方法。

注意，前三项从本质上不应该属于同一个地方——这是现代办公室设计最经常被完全忽略的原则，这一点将在下一节介绍。

9.6.1 一个可以和他人一起工作的地方

工作场所可以方便同事间进行交互。这样的交互分为两种：有计划的会见和自发的讨论。有计划的会见需要许多的会议室，每间会议室都应该有桌子、椅子、白板、电视或投影仪以及出色的隔音效果。对于自发的讨论，现代办公室大概有 70% 都选择使用开放式布局，大量的员工都坐在巨大的开放式区域中，要么共享办公桌，要么是在隔间里。这样的布局本来是为了改善沟通和产生点子，但大量研究却证明效果未必如我们所想。

例如在 1997 年的一项研究中，一家公司从传统的办公布局转变为开放式的布局后，就员工对物理环境、身体压力、同事关系和感受到的工作表现进行了评估，发现开放式办公室的每一项指标都更差，“员工的不满并没有减少，即便是经过了调整期之后”。丹麦在 2011 年的一项研究表明，开放式办公室可能会损害你的健康。和处于私人办公室的人相比，在开放式办公室工作的人要多请 62% 的病假。最后，2011 年对超过 100 项有关办公室环境的研究进行了评估，发现尽管“开放式办公室通常会营造出一种组织使命的象征意义，让员工感觉是在更加休闲和更有创造性的企业中”，但它们“对工作人员的注意力集中、工作效率、创造性的思维和满意度是有害的”。

其实“放松地交互”只是不到 10% 的工作者面临的问题，所以开放式办公只是没问题找问题的解决方案。这也没什么好惊奇的，这本来就是很自然的事，人们想要开始交谈，可以在办公室的任何地方，两三个同事就自然地聚在了一起，比如在厨房、饮水机旁、会议室或其他公共区域都可以。换句话说，我们很容易就可以设计出鼓励自发对话的办公室。另一方面，当你需要专注工作时，却很难设计出可以避免自发交谈以及避免工作被打断的办公室。

9.6.2 一个可以独处专注工作的地方

分心是专注工作的敌人。对于基本的脑力劳动（比如算法）来说，只要一小点办公室噪声都会产生影响。编程甚至需要更加集中的注意力，你必须把问题加载到大脑中，就像用纸牌搭房子，需要花时间并且耗费大量的脑力，而且一丁点儿干扰都可能把整个屋子弄倒，让你不得不从头开始，如图 9-7 所示。

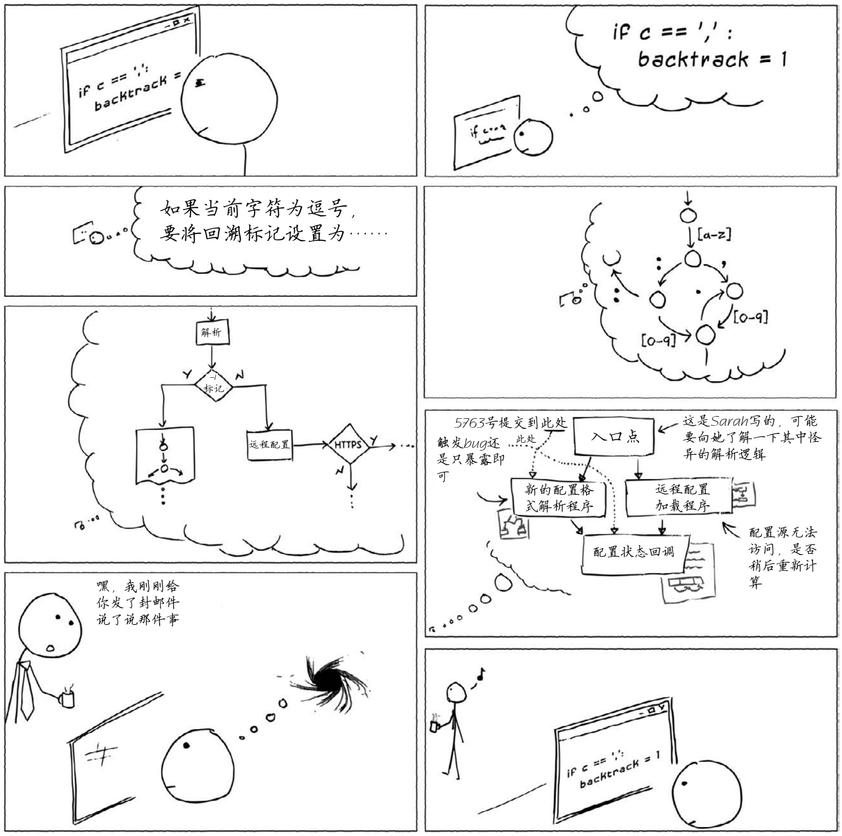


图 9-7：这就是不要打断程序员工作的原因

每一次大声叫嚷、每一次有同事拜访谈事情、每一次无用的会议、每一封 email 和每一通电话都是一种干扰。研究表明，一名程序员平均要花 10~15 分钟才能从干扰中恢复过来，重新开始编写代码。一个小时被干扰 4 次，就足以让你的生产效率下降到零。

> 这就是为什么搞开发的人为了回答你的问题，眼睛从屏幕离开时会给你一个如此不友善的眼神。因为他们脑子里的纸牌屋已经摇摇欲坠了。
>
> 仅仅因为存在会被打断的可能性，就会让这些开发人员不敢开始困难的项目。这也是为什么他们喜欢在深夜工作的原因，也是他们几乎不可能在隔间里写出出色软件的原因（除非在深夜）。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

开发人员的理想办公室就是他们可以心无旁骛地编写代码的场所。开放式办公室在设计上就容易让人分心，在很大程度上对程序员都是最糟糕的选择。更好的做法是为开发人员提供办公室——理想情况下是私人空间，但是对于小团队来说共享的办公室也是可行的。每间办公室应该有可以关起的门、出色的隔音和窗。窗是必需的，既是为自然采光考虑，也是为你提供一些东西，让你在深入思考的时候可以不光盯着显示器。

> 公司也希望每个人的办公室都能有一扇窗户，虽然我们知道那是不现实的。情况确实如此，最好的证明就是在一个空间中设置充足的窗户并不会增加过多的成本。现成的例子就是酒店——所有的酒店，你甚至无法想象，也无法忍受一间酒店的房间是没有窗户的（这还只是你睡觉的空间）。
>
> ——Tom Demarco、Timothy Lister，《人件》

办公室的成本有多少？当然，会比开放式隔间高，但可能不像你想得那么高。Fog Greek 是一家做项目管理工具的软件公司，估计出为每个开发人员提供私人办公室的成本大概是收入的 6%（注意，GrogGreek 位于曼哈顿，是世界上房产最昂贵的城市之一），这一数字也只是略微高于他们调查的其他类似公司。考虑到开发人员的薪水成本（通常接近创业开支的 75%）、生产效率以及办公室的幸福收益，私人办公室是物有所值的。这就是为什么很多软件公司都会为他们的程序员提供办公室，包括 Fog Greek 软件、CircleCI、SAS、Apple、微软、甲骨文以及 Google 的许多部门。

如果完全无法实现办公室，只能被迫接受开放式的布局，那最少也要为每一名员工配备噪声消除耳机作为补偿，并且还要实行"只有在紧急情况下才能打断戴着耳机的人"的政策。好的噪声消除耳机大约需要 100~200 美元，对于一个年收入 100 000 美元的开发人员来说，只要防止他在 2~4 个小时内不分心，就已经物有所值了。开放式办公室的另一选择是设置**码农洞穴**。这是一些专门进行安静、连续、专注工作的房间。码农洞穴可以是一个定制的房间（图 9-8 是 GitHub 的例子）、一间禁止谈话的变更了用途的会议室、一把完全封闭的椅子（例如泡泡椅）或者其他任何开发人员可以躲起来进行编码或思考的地方。



图 9-8：GitHub 的码农洞穴，由 Eva Kolenko 拍摄

9.6.3　一个可以放下工作的地方

有时候，你不在键盘边上反而可以将工作完成得更好。你是否曾经在清晨想到了前一天的问题的解决方法？是否在洗澡的时候突然灵光一现？是否在向别人描述问题时还没讲完就意识到了解决方法，即便其他人一个字都没说[4]？这些并不是巧合。研究表明，想要让大脑有效率、专注并具有创造性，我们需要让它休息休息。

举个例子，2011 年的研究表明，定期的休息可导致**目标二次激活**。也就是说，如果你离开工作简短地休息一下，当你回到工作中时，可以更好地退后一步，重新评估你的目标，看到大的景象而不是关注细微的实现细节。研究发现，这种方法还可以提升注意力和整体的工作表现。

想要通过短暂的休息来提高效率，你需要在开始感到筋疲力尽之前就休息——随机出现的干扰，正如前面提到的，并不是一种休息。例如，一些人使用番茄工作法，每 25 分钟就短暂休息一下，4 次重复之后又休息更长的时间。还有一些人喜欢工作 90 分钟休息 20 分钟这样的循环，和超日节律（Ultradian Rhythm）一致。我们不妨多尝试几种模式，看看哪种最适合自己。

这一切和办公室有什么关系吗？这就意味着休息应该成为工作文化中的一个固定部分。喝一杯咖啡、在饮水机旁聊聊天、做做锻炼，甚至浏览一下网页，这些都是合理的选择[5]。办公室应该专门设计，让这些互动可以在远离办公桌的地方进行，比如在厨房、自助餐厅、休息室、健身房，或者到户外区域走一走。这些方式比开放式区域能更好地激发不经意的沟通和想法的产生。

除了短暂的休息，较长时间的休息对于工作效率的保持也是必不可少的。每一个公司都有一名似乎无所不在的员工：早上 9 点出现在办公室，晚上 9 点还在 IRC 上聊天，凌晨 3 点又在提交代码。这样的人总是在工作。也许你就是这样的人（我知道我曾经就是）。你甚至会对自己长时间的工作感到自豪，认为自己是个英雄。但是更长的时间——每周超过 50 个小时，并不会提高工作效率。这一切只会增加压力、损害健康、让你犯下粗心的错误，最终情绪沮丧、逃离工作。换句话说，需要英雄事迹就是一种失败的信号，意味着一些事情严重缺乏规划和管理。

> 不需要英雄的环境好过一个团队都是英雄的环境。
>
> ——Ernie Miller，Nvisium 技术主管

当然，没有什么计划是完美的，每间创业公司时不时都需要一点英雄主义，但这不应该是工作的固定成分。如果偶尔的英雄成为了无所不在的英雄，他们就会因为工作而感到痛苦，他们的同事也是一样——为了不落后而感到压力，随即增加自己的工作时间。要避免这样的陷阱，我们需要营造一种文化，让员工不仅有灵活的工作时间，还应当是合理的工作时间。告诉员工要回家并保证他们的休假，必要的话还要强迫他们。例如 Travis CI 实施了一条**最少**假期政策。

注 4：这就是所谓的橡皮鸭调试，2.1.3 节有过解释。

注 5：研究表明，相对于根本不休息，休息一下、浏览网页可以提升我们的工作表现。

> 现在要求每个人每年最少休 25 天假（带薪），不管他在哪个国家生活。如果想休假超过这一时间，也没有问题，这也是最少政策允许的。但它只设置了较低的天数门槛，我们期望员工关注自己的幸福超过对工作的关注。
>
> 这条策略不仅仅是员工的指导方针，也是对每个人的命令，包括最初创建公司的人。作为领导，我们需要建立榜样，在工作和生活之间形成一种有益健康的平衡，而不是变成“生活就是忙于奔波”的榜样。
>
> ——Mathias Meyer，Travis CI CEO

强制人们不工作，而且还要支付薪水？如果这听起来很极端，可以想象另一种可能：**倦怠**。倦怠比疲劳更为严重，是一种精神和身体上一直都筋疲力尽的感觉。你经常在担心，每一件事情都令你不快，你也无法集中注意力。你会盯着远方，一次好几个小时，完全无法有效地做什么事情。你难以决定，无法入眠，人际关系也很糟糕。一般来说，一个人一旦倦怠了，他就会离开公司，无论给他鼓劲、晋升，还是加薪，都无法说服他留下来。记住，休假比离职补偿金和培训更加便宜。

9.6.4 一种可以根据个人需要布置办公室的方法

公司的每一栋房子、每一层楼、每一个部门如果都是一模一样的，给员工传递的信息就是公司仅仅把他们当作可替换、可交换的商品零件。如果你每年都要花几千个小时在办公室中，你会希望它感觉像家一样，而不是一间间相同的隔间组成的枯燥迷宫。你已经看到有很多研究都表明工作环境是有多么重要，但是我会再分享一点——能够控制你的工作环境同样也很重要。

> 2005 年有一项研究，范围涵盖了从美国中西部的汽车供应商到西南的电信公司。研究人员发现控制环境的能力对团队的凝聚力和满意度有显著影响。当工人们无法改变东西的外观、无法调整灯光和温度，或者无法选择如何召开会议时，他们的心情便会跌落谷底。
>
> ——Maria Konnikova，*The New Yorker*

我们应当让员工尽可能地控制办公室。例如，GitHub 维护了一个内部存储库，所有员工都可以在上面提出对办公室进行改变和改善的建议。如果对他们的建议感兴趣的人足够多，就可能得到落实。在 Linkedln，我们每隔几年就有一个“过道大改造”的比赛，每个人都会获得预算去装饰他们的隔间过道，根据大家的投票，装饰得最好的过道将赢得奖品。在这样的办公室里走动是很好玩的，你可以投票选出你的最爱，看看每个地方有多么特别。例如，我们的数据架构团队在它们的隔间旁边做了一面攀岩墙（见图 9-9）、IT 团队则把它们的整个办公区域变成了巨大的电子世界争霸战主题派对（见图 9-10）。

我们还应该让开发人员能够得到他们所需要的工具。如果你在小型创业公司，最好的选择就是让开发人员购买想要的硬件和软件，并给他们报销。如果你在较大的公司，可能是由 IT 部门管理所有的硬件和软件。那样的话，最好是每隔几个月进行一次调查，了解开发人员想要的东西。



图 9-9：LinkedIn 新的水平可扩展基础架构团队



图 9-10：IT 部门献出的电子世界争霸战主题，图片由 Mike Jennings 提供

开发人员通常会想要什么工具呢？下面是一个清单，你可以先了解了解：

- 适合编程的桌子（又平又大，能够调整高度）；
- 舒适的椅子；
- 运行速度快的笔记本电脑或台式机（最大的内存、CPU 和硬盘）；
- 一两个大显示器；
- 好用的鼠标和键盘；

- 办公室任何区域都有高速的互联网；
- 很多电源插座；
- 白版；
- 必备的办公用品（笔记本、便利贴、钢笔、马克笔、打印机）；
- 储物空间（可以放外套、包和私人物品的地方）。

9.7 远程办公

传统办公室的替代就是允许员工远程办公。许多公司都允许开发人员偶尔在家工作，但越来越多的**分布式公司**默认就是允许远程办公的。远程办公的员工占大部分的公司包括 37Signals、Automattic、GitHub、StackExchange、Typesafe、Mixcloud、Mozilla、TreeHouse、UpWorthy、Buffer 和 MySQL。这些公司大部分仍然拥有办公室，但他们愿意招聘世界各地的人，所以是否去办公室上班是可选的。

来看看这种方式有什么优缺点，以及远程办公的最佳实践是什么。

9.7.1 优点

不管你在世界的什么地方，从统计学上看，绝大多数程序员对你来说都是在其他地方。许多公司都说要招聘最好的员工，但如果你只招聘可以在本地办公室工作的应征者，你看到的只是现有程序员的极小一部分。分布式公司最大的一个优点就是，不用担心要找到恰好住在那里的开发人员，你能够从人数更多、更加多样的大量应征者中去招聘。

这是一种解放，让你能够接触到所有看起来大有作为的开发人员。这对于参与开源社区的公司来说特别有用。如果开发人员开始对你们的项目贡献代码，你就知道他们对你们所做的东西是感兴趣的，你们对其能力也有所了解，可以尝试把他们招入麾下，不管他们在世界的什么地方。

远程办公对于专注的工作也很理想。任何办公室其实都是一种一刀切的妥协。当你在远程办公时，你可以挑选完全适合你的环境。这也许意味着你可以在房间里的沙发上工作，也可以在咖啡店或者开放工作区工作，也可以坐在公园里工作。你可以看着窗外的风景，可以免于干扰，可以穿着你想穿的衣服，可以控制你的日程安排。这一点对于程序员是再好不过的，他们通常都是夜猫子；对为人父母者也很好，因为他们需要在白天照顾孩子。你甚至可以在工作的时候在全世界到处跑。下面是 Amazon 的一位工程师，他生活在船上。

> 大约 4 年前，James 和 Jennifer Hamilton 卖掉了他们的房子、汽车以及大部分世俗财产，搬到了（他们的船）Dirona 号上。现在，当停泊在西雅图的时候，Hamilton 会骑着自行车到 Amazon 的总部，通过 Amazon Prime 购物，在当地的 UPS 店面取走他的邮包。他无拘无束，有时候会乘着船到夏威夷并在那里工作。
>
> ——Robert Mcmillan，*Wired*

远程办公也可以节省花销。公司不需要支付办公区域和办公用品的费用，员工也不必浪费时间和金钱上下班，和小孩待在家的员工也许还能够节省日托和保姆的钱。

9.7.2 缺点

前文提到了办公室需要有 4 种东西：

(1) 一个可以和他人一起工作的地方；
(2) 一个可以独处专注工作的地方；
(3) 一个可以放下工作的地方；
(4) 一种可以根据个人需要定制办公室的方法。

远程办公对于后三项都是很理想的，但对于第一项“和他人一起工作”就有点麻烦了。在分布式公司中，大多数沟通都是通过文字进行的，比如 email、聊天工具、bug 讨论和 wiki。这种方式有优点，比如它是异步的，你不会打断任何人；你可以保留讨论的记录。但是它也有一些缺点，例如反馈的循环会比较慢。在办公室中，如果遇到了问题，你可以拍拍人家的肩膀，几秒钟内就可以得到答案（尽管这样也有缺点，因为你会让那个人分心）。采用远程办公的方式，你今天发送一封 email 或者在 IM 上发一条消息，可能要到第二天才会得到回复，特别是如果你们是在不同的时区。另外，文字相比较当面沟通来说是一种低带宽的沟通媒介，所以对于某些类型的讨论，比如决策制定、头脑风暴和反馈，它的效率是比较低的。

开会也比较困难。你可以使用视频会议，但如果参会的人分布在多个时区，找一个对所有人都合适的会议时间可能会很困难。即便你找到了好时间，每次会议开始时也至少会浪费 15 分钟的时间等人们都连接到会议中，捣鼓好他们的麦克风设置并努力让桌面共享起作用。即便都能好了，音频和视频也经常会不稳定，还时不时会掉线。这样的体验和面对面的交谈是不可比拟的。

但是最大的问题还是那种自发的、偶然的交互频率会大大减少。你无法和同事一起吃午餐，也无法一起运动、散步，或者下班后去当地的酒吧放松一下。所以想让几个人在一起开一个快速的头脑风暴会议会更加困难，想对新招聘的员工进行指导也更加困难，围绕一个共同的使命去凝聚员工也变得更困难，更难以传播共同的价值、在团队成员间构筑起紧密的关系，以及庆祝里程碑和成就的实现。一封“大家工作很出色”的邮件是无法和欢呼、举手击掌以及打开香槟时“砰”的那一声相提并论的。

> 远程办公的主要优点实际上也是主要的缺点。如果你只想关起门来心无旁骛地工作，真的也很简单，但是你也会失去很多内部沟通，因为你们并不是在相同的时间、相同的地点工作。人们在午餐聊天时脑海中浮现出来的许多想法也可能就不会有了。所以你需要用其他的方式去激发这样的行为，比如设置聊天室。
>
> ——Mat Clayton，Mixcloud 联合创始人

不是所有人都拥有在家工作所需的技能。你必须学会如何管理时间，如何让工作井井有条，如何通过文字高效地沟通，如何应对家人的干扰（例如小孩），如何在独处时保持专注和积极性。有些人在这样的环境下会茁壮成长，成为出色的远程员工，但有些人需要更多组织和人际上的互动才能取得成功。你也不想让你公司的私有数据存放在没有密码、全家共用、病毒滋生、通过不安全 Wi-Fi 和外部世界通信的家用电脑上吧。

9.7.3 最佳实践

> 这是一个二进制属性，团队要么完全是分布式的，要么就完全位于同一地点。我不信任远程员工，只信任远程（分布式）团队。也就是说团队中的每一个人，即便他们聚在同一条街上，即便他们两个人每天都去同一间办公室，即便他们坐在一间屋子里，他们仍然得像分布式团队那样来行事，确保所有的信息都以完全分布式的方式流动。否则你很容易会遇到这样的问题，两三个人在咖啡机前交谈，而忘了把交谈内容记下来让其他团队成员了解最新情况，这样总是会有人感觉被排除在外。这样是不行的。
>
> ——Jonas Bonér，Triental AB、Typesafe 联合创始人

“分布式团队”或者让信息以“完全分布式的方式”流动究竟意味着什么？最好的答案来自于 GitHub：你的团队应该像开源项目一样工作。开源项目天生就是分布式的，所以它们在过去 20 年形成的实践让这一模型运行良好。GitHub 也信奉这些实践以及应用的约束条件，并将其作为整个公司运行的方式。以下是从 GitHub 的内部产品开发文档中摘录出来的。

> 电子化：讨论、计划和操作过程应该使用高度精确的电子形式，比如 email、github.com 或者尽可能保留聊天记录。避免现实的讨论和会议。
>
> 可获得：工作应该是可见和可公开的过程。工作应该有网址，它应该能从某一件产品或某一次系统故障向后回退，并能够弄清楚情况是怎么发生的。最好是通过网址较短的媒介，比如 git、问题跟踪、pull 请求、邮件列表和有记录的聊天。
>
> 异步的：产品开发过程应当几乎没有哪一部分需要让一个人打断另一个人让他立即关注，或者要让他人在相同的时间出现在相同的地点，甚至在相同的时间出现在不同的地点。即便小的会议或者短暂的电话都可能打断工作，所以不妨考虑用（经过深思熟虑的）email 或发送 pull 请求的方式去安排这些事情。
>
> 不加锁：在设计的过程中要避免出现同步或加锁点，这在分布式版本控制系统上是显而易见的。我们没有让开发经理授权你提交代码到代码库之后才能工作，或者让发布经理去批准部署，或者让产品经理去批准开发实验性的产品构思。为实现目标的工作永远都不应该被审批阻拦。把批准或拒绝推到评审阶段处理，或者自动去处理，但不涉及核心的工作要很早就得到反馈。
>
> ——Ryan Tomayko，GitHub 软件架构师

除了拥抱开源模型，我所访谈的每一家分布式公司，包括 GitHub、Typesafe 和 Mixcloud，都为所有员工支付定期面对面会议的差旅费。这也是一个机会，公司可以让每一个人都认同公司使命、强化价值观念、加强公司文化，让员工可以面对面接触，彼此间保持紧密的联系。

> 我们让每个人一年可以乘坐 3 次飞机，每次 3~5 天。我们在几个办公地点之间轮转，像瑞典、瑞士和美国，每年在这些地方举办一次 Scala Days 大会。通常会有一次是强制性的，每个人都要参加，包括销售人员、市场营销人员和行政人员，其他两次则只是技术人员参加。人们都很喜欢这样的活动，他们真心感激。现在我们有 70 多个人，所以实际上挺花钱的，但是每一分钱都是值得的。我们做了很多事情，彼此也相互认识。这可能是最重要的。

> 独立的团队也会更定期地见面。Akka 团队每隔一周会见一次。他们全都在欧洲，会飞到欧洲的某个地方见面。而 Play 团队则每隔一个季度见一次。他们可以更频繁地见面，但还是决定不这样。所以，对团队来说怎么样都可以。当然，我们也有预算，但预算是非常有弹性的，因为见面太重要了。
>
> ——Jonas Bonér，Triental AB、Typesafe 联合创始人

> 当你了解 Mixcloud 团队的分布情况后，你就知道他们中有很多人都是在离伦敦一小时的距离内，所以产品团队中有很多人都会相当有规律地见面。如果我们有某件工作是多个人参与的，我们就会让所有人集中到一个地方两三天。对于在罗马尼亚的伙计，我们一个季度要飞过去一次。他们也会过来花上一个星期的时间和当地每个人在一起。后来这几年我们让整个产品团队出去度假一周，这不是真正的假期，实际上更多的是不停地工作。我们租下几间别墅，让所有人都在同一屋檐下，我们从早上 9 点工作到下午 5 点，5 点之后，我们就出去找乐子去了。
>
> ——Mat Clayton，Mixcloud 联合创始人

最后还有一个建议：大多数分布式公司开始时都是本地的公司，联合创始人通常彼此间都认识，居住在同一个地方，前几个月都在同一间办公室工作（阅读 11.2.1 节了解更多信息）。在创业的初期，让所有人都在同一个地点工作是很有帮助的，因为那时候仍然要尝试确定公司的文化，并寻找产品的市场地位。只有在清楚公司**是什么**并且把关注点转移到想出**如何做**之后，我们才应该考虑通过招聘远程员工的方式去扩大团队规模，成为一家分布式的公司。

9.8　沟通

每一次对话、每一封 email、每一份 bug 报告、每一个 wiki 页面、每一个口号以及每一次压力的释放，都会反映和塑造公司的文化。沟通就是你如何传播你的文化以及如何去改变它。本节将更细致地介绍两种沟通的类型：内部沟通（员工彼此之间进行沟通的方式）以及外部沟通（员工与外部世界进行沟通的方式）。

9.8.1　内部沟通

沟通的第一个问题肯定是应该说什么和不应该说什么，换句话说，就是你的公司有多么“透明”。你们乐意分享的程度取决于你们的核心价值以及所从事的业务类型，但大部分的创业公司应该要尽可能的“透明”。例如，HubSpot 的其中一个核心价值就是要做到“极端明显的透明”。在内部，他们努力将所有可以分享的东西和员工进行分享。在他们的内部 wiki 上，员工可以找到公司的财务状况（现金余额、资金消耗率、损益表等）、董事会议的幻灯片、管理会议的幻灯片、有关公司“战略”的内容，以及许许多多有趣的 HubSpot 传说和神话页面。

有几样东西是几乎每一家公司都应该在内部分享的，这样才能让所有员工都目标一致地推动公司的前进。举个例子，我们不妨试试这个实验——到你的公司问每一位员工：“公司今年优先要做的前三件事是什么？”如果你从每个人口中得到的答案不同，你就需要在内部沟通上进行一些改进了。让员工了解公司优先考虑的事情、了解公司正从事什么

项目去实现这些优先考虑的事情、过去有什么项目成功了或者失败了、了解公司的财务表现，这些都应该是可以轻易做到的[6]。此类信息应该可以在内部的 wiki 网站或仪表板中获取，而且应该把更新定期发送给所有员工（例如以季度报告的形式）。有一些信息，比如公司的使命和价值，甚至应该公开发布出来，就像 Netflix 的 Culture Deck 和 HubSpot 的 Culture Code，这些内容在 Slideshare 上都有数百万的观看次数。

除了 wiki 和仪表板之外，**全体大会**也是内部沟通的良好工具。你应该至少每个季度开一次全体人员大会。在 Linkedln，我们每两周开一次全体大会。我们让所有人进入同一间房间（当公司规模扩大时，则要进入同一个视频会议），CEO 将会讨论公司优先考虑的事情、庆祝我们的成功，以及，也是很重要的，从失败中学习。

> 健康的公司文化鼓励员工去分享坏的消息。可以自由、公开讨论其问题的公司也可以快速地解决这些问题，而掩盖问题的公司则会打击员工参与的积极性。所以 CEO 的做法应当是：营造一种文化，奖励（而非惩罚）人们将问题公开，使得问题得以解决。
>
> ——Ben Horowitz，《创业维艰》

在全体会议期间，会有一两个团队有机会出来让公司的人知道他们正在做什么。CFO 也许也要站出来和大家分享公司的收益数字，或者让产品经理告诉我们即将使用的新设计，或者让工程师谈谈我们已经开源的技术。我们还有一个传统，就是让新招聘的员工在全体大会开始时，向整个公司做个自我介绍，必须告诉大家自己的名字，在 Linkedln 从事什么工作，一些没有放在 Linkedln 个人资料中的信息，然后还有一个有趣的环节：你必须展示一项特殊的技能、天赋，如果什么也没有，你可以学一种动物的叫声。我见过有员工唱歌、演奏乐器、表演魔术、即兴说唱、表演运动特长，还有模仿猫、狗、马和唐老鸭的声音。甚至还有一组实习生打断了 CEO 的讲话，表演了一段精心编排的舞蹈快闪。

全体大会还有另一种替代的形式，就是定期的公司宴会。

> Twilio 是一家 API 公司。我们有一个传统，要求每一名员工必须实现并演示一个应用，这个应用必须是用 Twilio API 实现的，为此可以得到有公司 logo 的外套和 Kindle（这是一项不受限制的公司福利），适用于所有部门，包括技术、销售、财务和市场。
>
> 我们每个周三都有一次公司宴会，新员工会在宴会上演示他们的应用，而我们的 CEO 则会把外套给他们穿上，为他们“授爵”。当新员工展示他们所做的东西时，看到整个公司都为他们喝彩是很美妙的，不管他们做出来的东西多简单或多复杂。
>
> 对于大多数非技术员工来说，他们演示的应用是第一次完成的软件开发。为了支持他们完成，我们有一个工程师固定在每个星期做一次下班之后的代码辅导，任何人都可以参加并获得帮助。
>
> ——Renee Chu，Twilio 软件工程师

注 6：对于上市公司则是例外，上市公司披露财务数据需要严格遵守证券委员会（SEC）的条例。

有些创业公司制定了一些政策，针对员工彼此间应该如何进行沟通。例如 GitHub 发布了 **GitHub 的 15 条沟通规则**，尽管这更多的是偏好而非规则。这其中包括更倾向于使用异步沟通（“网络聊天天生是异步的，拍拍别人的肩膀则是一种让人难受的行为”）；使用问题跟踪程序作为处理大多数问题、想法和 bug 的方式；除了敏感的对话以外，偏向于使用问题跟踪程序或者聊天工具取代 email 去处理所有的事情（“email 通常只用来处理个人讨论、一对一反馈和外部沟通这样的事情”）。支付创业公司 Stripe 对 email 的态度则稍有不同：按照约定，Stripe 的每一封邮件都会被抄送到整个公司或者一个特定的团队。尽管这些 email 需要进行大量的过滤，但有助于公司保持一种开发性，让所有人都可以轻易了解公司正在做的事情。

9.8.2 外部沟通

和内部沟通一样，外部沟通的第一个问题也是你应该和不应该说什么。有些公司专门雇了 PR（公共关系）团队，限制员工与外部世界的沟通，严格控制它们的形象和品牌。有些公司则采取了相反的做法，鼓励每一个人尽可能去分享。Buffer 是一家做社交媒体管理工具的创业公司，以最彻底的透明化为信仰。他们公开分享与公司有关的几乎所有一切，包括每名员工的薪水和股权方案、投资人的协议条款以及公司的所有指标，包括财务的详细分析。

如此透明的好处是它产生了信任，薪酬不再是员工之间小心提防的秘密。因为每个人都可以获得同样的信息，玩弄权谋的空间更小，没有人会感到他们可能被压榨。此外，公司对客户、投资者和员工也更负责任，任何的歧视、不公或者不道德的行为都会更加明显。在 Buffer 公开薪酬信息之后的那个月，他们收到的简历是正常情况下的两倍，应聘者中的文化契合也要高得多。

彻底的透明在其他创业公司中可以找到，比如 Moz、SumAll、Semco 和 Balanced Payments，而在创业领域之外也同样存在。例如，每一家上市公司根据法律要求必须在每个季度披露他们的产品指标、财务业绩以及管理层薪资水平。许多政府雇员的薪酬信息也是公开的。另一个著名的例子是职业运动，举个例子，如果你对美国职业棒球大联盟感兴趣，有一个网站可以让你查询每一名球员的薪酬以及每一支队伍详细的估值分析。

在找到了你们所适应的透明程度之后，有三种主要的外部沟通类型可供考虑：第一是如何设计、营销和推广你的产品，这点在 4.2 节已经讨论；第二是如何通过博客、开源软件和展示去宣传自己的公司，这一点将在 12.2.3 节讨论；第三是如何和客户沟通，这点已经在 4.2.1 节讨论过。

9.9 过程

如果公司的核心思想是**为什么**，那么过程就是**怎么做**。虽然“为什么”应该尽早定义并在整个公司一以贯之地执行，但是“怎么做”却要尽可能长时间地留给每一个人去决定。我们招聘有才能员工的全部原因就是他们很专业。最接近问题的人就是能够弄清楚问题解决过程的最佳人选。自上而下施加的死板的行动决策通常只会增加开销并剥夺个人的自主权。换句话说，如果我们已经很好地定义了为什么，怎么做的问题自然就不用操心了。

> 为什么文化对商业来说如此重要？其实不难理解，文化越浓厚，公司所需要的共同执行的过程和步骤就越少。当文化很浓厚的时候，你可以信任每个人都会做正确的事情。人们可以做到独立自主……你有没有注意过家庭或者部落并不需要多少过程和步骤？那是因为其中存在强烈的信任和浓厚的文化，取代了任何的过程。在文化薄弱的组织（即便是社会）中，才需要各种各样严密的、精确的规则和过程。
>
> ——Brian Chesky，Airbnb 联合创始人、CEO

如果在文化上做好了，过程只要做到一点即可：采用出色的判断。

9.9.1 采用出色的判断

Hubspot 并没有一本厚厚的政策和规程手册。相反，他们对每件事情都使用了一条 3 个词的政策：采用出色的判断（use good judgment）。

> - 社交媒体政策。
> - 差旅政策。
> - 病假政策。
> - 活动的饮料购买政策。
> - 暴风雪期间在家工作政策。
>
> 我们对所有这些（以及其他大多数）事情的政策就是：采用出色的判断。
>
> ——HubSpot Culture Code

Nordstrom 在它的“员工手册”（即单张 5 × 8 英寸的卡片）中也表达了一条类似的理念。

> **欢迎来到 Nordstrom**
>
> 很高兴你来到我们的公司。
>
> 我们的首要目标是提供不同凡响的客户服务。请设定远大的个人和职业目标，我们对你实现目标的能力有充足的信心。
>
> **Nordstrom 规则**
>
> 第一规则：在所有情况下采用你的出色判断
>
> 除此之外没有其他的规则。
>
> 请在任何时候自由地向你的部门经理、商店经理或者分部总经理提问。
>
> ——Jim Collins、Jerry I. Porras，《基业长青》

这并不意味着完全不采取任何过程，但是“采用出色的判断”应该是默认的过程。对于特定的情况，只有在“出色的判断”被证明不够时，才有必要在此之上实施一些额外的步骤。比如说，假设一名开发人员在修改数据库配置文件时不小心引入了一个 bug，这个 bug 在半夜让数据库停了下来，你必须争分夺秒地去修复它。处理这个问题有 3 种方法：

(1) 讨论并记录下哪里出错了，然后继续你的生活；
(2) 实现一个自动化的解决方案，防止该问题以后再出现；

(3) 引入手工的过程，防止该问题以后再出现。

大多数情况下，第一个选项是合适的选择。讨论什么地方出问题可以让你有机会从错误（或者你同事的错误）中学习，可以将你“采用出色判断”的技能提升到足以保证此类 bug 永远不会再发生。用文档记录下问题可以让所有人从之前的错误中汲取教训，也有助于找出“采用出色的判断”不足以解决问题的特殊情况。例如，如果你在文档中看到数据库配置的 bug 已经发生过许多次了，可能就不仅仅只是讨论和记录的问题，还要找到更好的解决办法。那样的话，下一步应该就是寻求第二个选项——自动化。

计算机比人更擅长执行严格、重复的过程，所以一旦你找到的问题无法通过单独的判断去解决，就应该尝试建立自动化的解决方案。自动化测试（阅读 7.2.2 节了解更多信息）、静态分析（阅读 7.2.3 节了解更多信息）、持续集成（阅读 8.3.3 节了解更多信息）以及持续交付（阅读 8.4.3 节了解更多信息）可以防止很大一部分问题的出现。例如，我们也许希望能够将每一次数据库配置的修改自动部署到准生产环境中，并对它运行自动化测试，从而在数据库配置的 bug 进入生产环境之前能够捕捉到它。只有在解决方案实现自动化不切实际的情况下，我们才应该考虑第三种选择——手工过程。

引入手工过程应该永远都是最后的手段。例如，我们可以要求开发人员每一次修改数据库，都必须填写一张“数据库申请单”，交由 DBA 批准；也可以要求开发人员每一次部署都填写“部署申请单”，由发布工程师批准。问题是增加这样的手工过程的成本是昂贵的。公司以后所有数据库和部署的更改都会增加额外的开销，其实也是在剥夺开发人员的自主权，让他们无法从错误中得到学习，把它们当成需要定期借助外部机械力量去完成任务的机器人。你虽然付出了这样的成本，却不能保证可以得到更好的解决方案。

许多手工的过程都是无效的，因为它们依赖于人去执行重复的任务，而这正是我们不擅长也容易厌恶的事情。配置的 bug 完全有可能从 DBA 或发布工程师的手中溜走，正如它从原来的开发人员手中溜走一样。此外，这样的问题可能不常出现，所以手工过程的开销对公司带来的成本远比在问题出现的时候再进行修复要高。换句话说，有些类型的 bug 不常出现或者解决成本非常低（阅读 8.3.3 节了解更多内容），所以在这些 bug 出现的时候再做反应会比试图防止它们出现效率更高。

9.9.2 软件方法论

但是软件开发的方法论又如何呢？它们能够阻止数据库 bug 这样的粗心错误吗？我们应该使用敏捷、瀑布、XP、Spiral、Crystal、BDD、FDD、DDD、DSDM、PDD、精益、Scrum、Kanban 或 Scrumban 这样的过程吗[7]？有些过程对大型项目和团队的组织是有作用的，但不能期待它是银弹。

> *没有哪一种开发，无论是技术还是管理技术，承诺 10 年之内能在生产效率、可靠性和简单性上哪怕提高一个数量级（10 倍）。*
>
> ——Fred Brooks，《人月神话》

在从事大型项目开发时，复杂性分为两种：内在复杂性和偶发复杂性。内在复杂性是我们尝试解决的问题本身就自带的。举个例子，如果你在开发金融交易算法，找出能够战

注 7：其中有的方法论是我捏造的，看看你能不能找出来。

胜市场的算法就是该问题本身复杂性的一部分——没有办法绕开它。偶发复杂性则是该问题偶然引起的，或者在解决问题时所采用的某种特定方法的副作用所导致的。如果我们使用 C++ 去实现金融交易算法，那么内存泄漏和段错误就是一种偶发复杂性，源于使用该语言不得不进行手动的内存管理——我们可以选择具有自动化内存管理技术的其他语言（比如 Java）来避免这种复杂性。

内在复杂性在部分软件项目中占大多数，也没有哪一种软件方法论可以帮助我们避开它。我们最多寄希望于软件方法论能让偶发复杂性降到最低——比如两个同事由于糟糕的沟通而把时间浪费在一些重复的工作上——但也不要期望能有多大的效果，因为这些方法论之间的差异通常很小。换句话说，开发人员之间的差异可能是巨大的（阅读 11.2.4 了解更多信息）。我们招聘的人（**谁**）以及让他们和共同的使命保持一致的方式（**为什么**），相比于我们挑选的方法论（**怎么做**），前者对项目的成功有更大的影响。

> 大部分的组织都存在令人恼火的事情，它们的好坏取决于招聘到的人。如果我们能够绕开这种天生的限制，即便组织所雇佣的是普通或不称职的人，也能一样的出色，这不是很好的事情吗？没有什么比这更简单——我们所需要的（此处应有掌声）只是一种方法论。
>
> ——Tom Demarco、Timothy Lister，《人件》

这一切都意味着，公司引入软件开发方法论和引入其他类型的过程并没有什么不同——都是在“采用出色的判断”已经被证明不够了才采取的措施。我采访过的大部分创业公司都避免使用那些有着花哨名称、有专门介绍的书和认证的方法论。但是当他们发展到足够大之后，差不多全部都会遵循一种过程，这种过程用“近似敏捷”来描述再贴切不过了。

为什么是敏捷？这本书的其中一个关键主题就是成功的公司是进化的结果，而不是天才的设计。这个思路与**敏捷宣言**吻合度很高。该宣言声称，“响应变化”比“遵循计划”更有价值。敏捷鼓励一种定期从客户获得反馈的迭代式和增量式的开发过程（阅读 2.2.1 节了解更多信息），也许最重要的是敏捷过程有一个关键的因素：它能够进行自我进化。大多数的敏捷方法论都包括定期的**回顾会议**，以评估这一过程中哪些是有效的，并在必要时采纳它以满足组织的需求。有关如何在自己的公司中建立敏捷过程的实践指南，请阅读《解析极限编程（第 2 版）》及 *The Art of Agile Development*。

为什么只是“近似”敏捷？因为任何方法论如果被太过严格地应用，都会存在缺点。大多数软件方法论的一个目标就是“提升最低水准”。也就是说，它们会强加一种故意不能变通的过程，使得常见错误的出现减小到最低程度，实现输出一致性和可预测性的最大化。不利的一面则是，编程天生就是一种创造性的工作，所以任何不可变通的方法本身就会抑制这样的创新以及创造性地解决问题。换句话说，严格地应用一种方法论去提升最低水准是以削弱最高水准为代价的，它会阻碍开发人员的最佳表现，也不利于形成不断改进的文化。

9.10 小结

《基业长青》一书对一项 6 年研究项目的结果进行了概括，该项目致力于研究成功打造一家“有远见的公司”的必要条件——被认为是所在行业中最出色的公司之一。这样的公司

历经多年，产品和领导也经受住了考验，对世界产生了持续的影响，比如迪士尼、IBM、波音和通用电气。《基业长青》中有一个关键发现，就是有远见的公司的领导会关注创立伟大的组织而不是伟大的产品——那样的公司只是“制作钟表的”，而不是“时代的讲述者”。他们最伟大的产物并不是特定的想法或产品，而是公司本身及其代表的东西。

换句话说，伟大的公司起源于伟大的文化，它们基于一个激动人心的使命而建立。创立者定义了公司的核心价值，并把这些核心价值转化为每一个决定。他们通过组织设计、招聘、晋升、激励、办公室、远程办公、沟通和过程等方式，让公司与使命、价值保持一致。而他们所做的这一切并不仅仅是为了创造伟大的产品，或者为了赚到许许多多的钱，而是为了建立一家伟大的公司。

> “公司文化”并不能脱离公司本身而存在：没有哪个公司是拥有一种文化的，每个公司就是一种文化。创业公司是肩负使命的一队人马，好的文化其实就是其内在的体现。
>
> ——Peter Thiel，《从 0 到 1》

Tom Preston-Werner 是 GitHub 的联合创始人之一，他在一次演讲中曾经问过“为什么公司会存在”。是为了赚钱，还是为了让人们快乐？他使用了一种“减法证明”来支持后面一种观点。

> 一家公司减去利润，得到什么？创业公司。一家公司减去人，得到什么？一无所有。它已不复存在。
>
> ——Tom Preston-Werner，GitHub 联合创始人

这是一个有趣的证明。但在这次个名为“Optimizing for Happiness”的演讲中，有一个严肃的观点：公司应该是让人快乐的，而不是获得利润。换句话说，创业是与人密不可分的。为快乐而优化的创业公司可以产生一种良性循环：当员工感到快乐时，他们会让公司变得更加强大；当公司强大时，它又使员工更快乐。所以世界上最成功的创业公司都不遗余力地用本章所介绍的一些稀奇古怪的文化特色去让员工高兴，比如 GitHub 的美国总统办公室、Google 的“20% 时间”和 Twilio 的“授爵”仪式。其中有些实践也许听起来很疯狂、很费钱或很浪费，但却很好玩。老实说，你难道不是更喜欢在这样有趣的公司工作吗？

> 再也没有理由相信只有无意义的玩耍才是享受，生活的要义也未必是要背负着十字架艰难地前行。只要我们意识到工作和玩的边界其实是人为的，我们就可以处理好这些事情，开始努力让生活变得更有价值。
>
> ——Danniel Pink，《驱动力》

第 10 章

求职之路

阅读了第 1 章之后，你可能想去创业公司工作了。那么，如何才能找到一家好的创业公司加盟呢？如何才能让他们对你感兴趣？在面试的时候你要怎么做？如何对公司及其职位进行评估？

找到好工作的第一个步骤就是让自己变得出色，这一点将在第 12 章讨论。本章会带着大家经历求职的全过程。我将先讨论如何找到创业公司的职位，然后再探讨想要在面试中胜出需要做些什么，最后讨论如何对工作机会进行评估和谈判。

10.1 寻找创业公司的工作

找到在出色创业公司工作的机会的第一步就是知道自己在寻找什么。我们不妨坐下来试着问自己下面这几个问题。

你关注什么行业和什么类型的产品？

　　例如：医学、电子商务、新闻、旅行、社交网络、游戏、视频、金融、通讯、安全。

你对什么技术充满热情？

　　例如：嵌入式系统、移动应用、分布式系统、函数式编程、机器人、生物传感器、机器学习、信息检索、图形技术。

你对什么类型的商业模型感兴趣？

　　例如：广告（展示、联盟、商机引导），商业（零售、市场、拍卖），订阅（SaaS、会员、付费），个人对个人（消息传送、分享、购买），交易处理（商家、银行），数据（商业智能、市场研究），开源（咨询、支持、托管、许可）。

你在寻找什么样的职位？

　　例如：入门级、高级、领导者、经理、CTO、前端、后端、工具、开发运维。

还要考虑哪些其他因素？

　　例如：位置、通勤、公司规模、差旅、免费食物、家中工作。

这些问题的重要性对你来说各不相同，而且某些问题你也没有答案，特别是如果你还比较年轻的话。然而，写下你知道的问题并加以关注，会帮助你有针对性地搜索工作岗位。

10.1.1　利用人脉

> 机会并不会像浮云一样飘走，它们会牢牢地与个人联系在一起。如果你在寻找一个机会，实际上是在寻找人；如果你在评估一个机会，实际上是在评估人；如果你尝试统筹资源去追逐一个机会，实际上是尝试得到他人的支持并参与其中。并不是公司给你提供工作，工作是人提供的。
>
> ——Reid Hoffman、Ben Scanocha，《至关重要的关系》

现在你对寻找的工作已经有了一些想法，但是先别迫不及待地开始在网上的招聘版块上细细寻觅。有 80% 的工作并没有公开刊登广告，找工作的主要时间应该花在和你的人脉的互动上。去找你的朋友和同行交谈，让他们知道你正在找什么工作（根据自己对上一节问题的回答）。他们会告诉你一些不为人知的工作机会，相比于招聘广告，他们会为你提供更加真实的工作情况。他们的引荐也会提升你被录用的概率，通过工作版块得到录用的概率大概是百分之一，而通过引荐被录用的概率则是七分之一。

那么，如何才能与合适的人取得联系呢？如何构建自己的人脉呢？你的人脉也许比你想象得更广阔。Linkedln 就是一种审视人脉的强大工具，它不仅包括你的个人关系（第一度），还包括他们的关系（第二度），以及他们的关系的关系（第三度）。只要你有 50 个有关系的人，每个人又有 50 个关系，以此类推，你第三度的关系人将达到 125 000 人（50 × 50 × 50）。例如，如果你访问感兴趣的公司的页面，你就能够在右上角看到与你有关系的人，如图 10-1 所示。

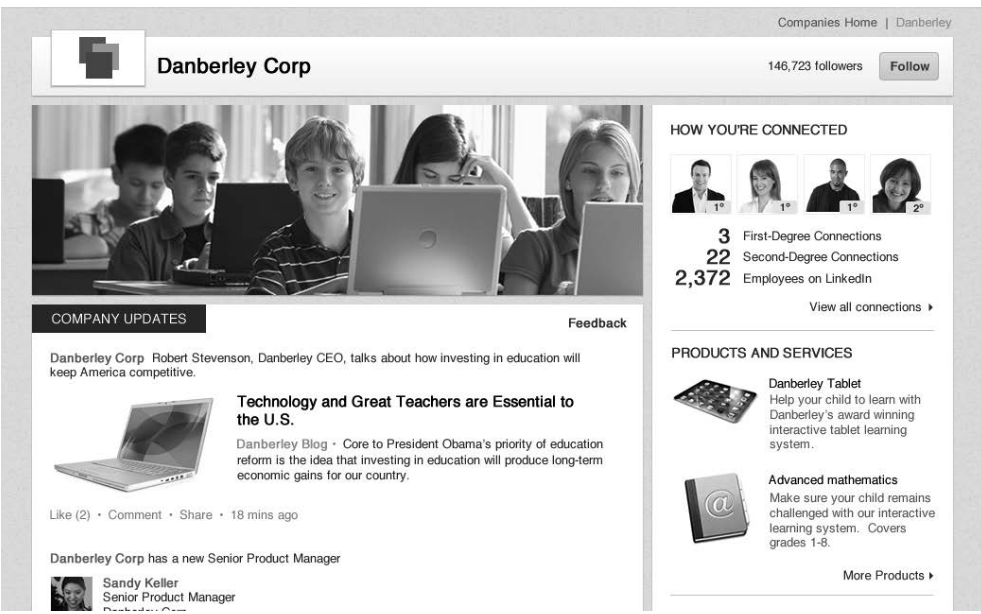


图 10-1：Linkedln 的公司页面展示了你与该公司之间的联系

如果你已经具备了第一度的联系，就可以给他们发送消息。如果你看到的是第二度或第三度的联系，可以点击他们，查看他们的资料。在资料页的右侧，可以看到你和这个人有什么样的联系，你们有何共同点，如图 10-2 所示。



图 10-2：Linkedln 的个人资料页面展示了你与某个人之间的联系

你也可以在 Linkedln、Facebook、Twitter 和 email 地址簿中浏览你的联系人，向他们请求介绍、引荐的机会，或者干脆邀请他们去喝咖啡聊天，也没必要太过正式。大多数人都喜欢谈论他们自己和他们的工作，所以只要告诉他们你对他们从事的工作有兴趣就可以了。如果有职位空缺，在交谈中他们自然就会让你知道。

10.1.2 发展人脉

如果你还没有找到合适的机会——也就是合适的人，那么接下来你优先要做的就是发展你的人脉。如果想认识新的人，你必须去新的地方。

1. 聚会小组与会议

你对 Java 感兴趣吗？截至 2014 年，Meetup 网站列出了 66 个国家的 900 个有关 Java 的聚会小组，会员总数超过了 280 000 人。想要捣鼓一下 Node.js ？上面有 386 个小组，遍布 54 个国家，共 94 000 名会员。大数据让你兴奋？有位于 72 个国家的 2156 个小组可供你选择，还有超过 50 万名数据科学家会加入其中。如果你想参加比聚会小组规模更大的活动，世界各地也有大量的会议：2013 年，Lanyrd 登出了 153 个 Java 会议、125 个 Node.js 会议和 232 个大数据会议。

聚会小组和会议在什么地方都有，而且这是学习新技能和认识社区新人的很好的方式。他们也提供了大量机会，让公司为它们空缺的职位做广告。许多聚会小组在开始时都是招聘宣传，大多数会议赞助商、组织者和演讲者也都会为一些工作机会做宣传。

如果你无法在本地找到好的聚会小组，可以自己开始组织！不需要非得很大的规模，只要宣传一下，聚起 10~20 个开发人员去讨论你感兴趣的主题就可以了。这样你不仅有机会学习新的东西、认识新的人，也能够建立起自己作为聚会组织者的品牌。

2. 黑客马拉松和比赛

编程比赛是提升技能、认识新人、获得免费食物与奖品，以及赢得现金大奖的好方式。几乎每一次编程竞赛实质上都是一次招募活动：公司（即赞助商）会在这些活动上出钱，换取招聘到参赛者的机会。

编程竞赛有许多变种，包括算法挑战赛（让所有人都面对完全相同的问题和约束，目标就是找出解决该问题最有效率的算法）、数据挑战赛（给你一个数据集，让你提出最有趣的见解或者预测）以及黑客马拉松（一种开放式的竞赛，你可以生成最佳的解决技巧，炫耀你的创造力，当然有时候也会限制在某个特定的主题，例如改善教育的黑客技巧）[1]。

> 我曾经遇到过一个人，他是一家银行的软件工程师（不是 JP 摩根或者其他顶级的银行，只是中等的、相对不那么知名的区域银行），在两三年前给我发了电子邮件，说到“我想加入一家创业公司，应该做些什么呢”。我的回答是，出去参加两三次黑客马拉松，花几个星期去做项目。你可以把你的简历水平从 B 或 C 变成至少 B+ 或 A−，这一点在几个月内就可以做到，然后再去硅谷，申请公司职位就可以了，因为湾区或任何科技中心的机会要比其他城市的机会多得多。他在一年后发邮件给我，告诉我他已经成了 Uber 早期的工程师，赚到了许多钱，现在的情况好了很多。
>
> ——Gayle Laakmann McDowell，CareerCup 创始人、CEO

读者不妨访问 http://www.hello-startup.net/resources/jobs/ 网站找找你周边的编码竞赛。

3. 演讲、博客和开源

如果想让社区关注你，最好的方法就是为社区做贡献：做演讲、写博客、将代码开源、处理邮件列表、在 Stack Overflow 上回答问题、在 IRC 上聊天。只要我参见会议或聚会小组，离开时便可以获得 5~10 个新的联系人；只要我在会议或聚会小组上做展示，离开时便可以获得几十个新的联系人。读者可以阅读 12.2.3 节，其中对撰写博客、演讲和开源有全面的讨论。

10.1.3　创建网络身份

如果不能通过人脉找到工作，那么退而求其次，看看能不能让工作找到你。在过去的 5 年，我已经收到大约 1300 封电子邮件，这些邮件来自创业公司的创始人、招聘经理和招聘人员。我的收件箱在几乎每个工作日都会收到至少一个工作机会，我不再需要申请工作——而是工作向我申请。

几乎每一名程序员都可以在恰当的地方创建一个网络身份，从而实现类似的事情。如果想让公司找到你，你的名字应该出现在它们会去的地方。第一站就是 Linkedln。我作为 Linkedln 的前员工肯定是有私心的，所以我说的不算。但在 *The Recruiter Honeypot* 一书中，Elaine Wherry 讲述了她如何尝试用 Pete London 的名字，伪造成网上一个“JavaScript

注 1：这里使用的是“黑客”（hack）一词原始的定义，即为了乐趣而编写计算机程序或者以不那么优雅的方式去拼凑或临时做出某种东西，通常作为快速原型或者概念的验证。这个词在这里不是现在新闻媒体上用的意思，后者指非法入侵到计算机系统和网络中。

忍者”式的人物去吸引招聘人员的注意。

> 我酝酿着一个想法，想用一个编造的名字把我的简历放到网上几天。在一个不眠之夜，我起床弄了个三个页面的小网站，分别是“关于页面”“简历页面”和“博客页面”。这个网站是 Pete London 的，他是一个假设出来的人，兴趣和技术经历除了不是创始人之外，都是照着我自己弄的。我把我毕业前的经历调换成我丈夫的，这样就不会太容易追溯到我。这一切都没有告诉任何人，我怀着一线希望回到床上——我搜寻招聘人员已经好几个月了，但这回招聘的人将会反过来找我！
>
> 我的希望很快就落空了。PeteLondon.com 孤零零地在互联网上待了几个星期，没有任何的活跃度。我正打算将整个网站拿下来时，又想了想，最后再把简历发到 Linkedln 上试一试。
>
> 砰！就好像最后偶然发现了通往派对的大门。在 2009 年 12 月 10 日，第一条 Linkedln 消息来自于 Google，Mozilla 的消息又在 12 月 15 日紧跟而至，Ning 和 Facebook 的消息又在 1 月接踵而来。从此之后，Pete 平均每 40 小时就会收到一条招聘信息，总共收到了 172 个组织 382 位招聘人员的 530 封 email。
>
> ——Elaine Wherry，Meebo 联合创始人

Julia Grace 在“Tips for Finding Software Engineering Jobs”中也有同样的说法。

> 错误：人们真的会使用 Linkedln 吗？我不这么认为，我最后一次更新个人资料是两年前的事了，也没有人去看它。
>
> 正确：人们一直在使用 Linkedln。真的，一直都是这样。它是招聘人员的利器……公司负责面试的人也许也会使用——相对于简历，他们更有可能阅读你的 Linkedln。（点击链接和打开一份 PDF、Word 或文本文档相对比，你会选择哪种方式？）保持信息更新并让人们知道——比起 PDF、Word 或文本简历更容易维护、更容易被找到，甚至也不必自己去托管！我已经有 4 年多没有使用过 Linkedln 之外的简历了。
>
> 底线：你必须要让新的工作机会可以很容易地找到你。在职业生涯早期，还没有人知道你是谁，甚至不知道你的存在时，这一点尤为正确。假设一年前你见过的一位熟人在一家火热的新创业公司工作，想了解你是否对加入进去感兴趣。他要做的第一件事情肯定是搜索 Linkedln。
>
> ——Julia Grace，WeddingLovely 联合创始人、Tindie CTO

如果想创建有大量点击的 Linkedln 个人资料，你需要弄明白招聘人员和招聘经理是如何使用 Linkedln 的。大部分人都在使用一款叫 Linkedln Recruiter 的产品，通过关键字进行搜索和布尔查询，比如用“Java AND JavaScript AND MySQL”这样的语句。他们也会对工作年限、学位、研究领域、之前的公司和所在地进行过滤，如图 10-3 所示。

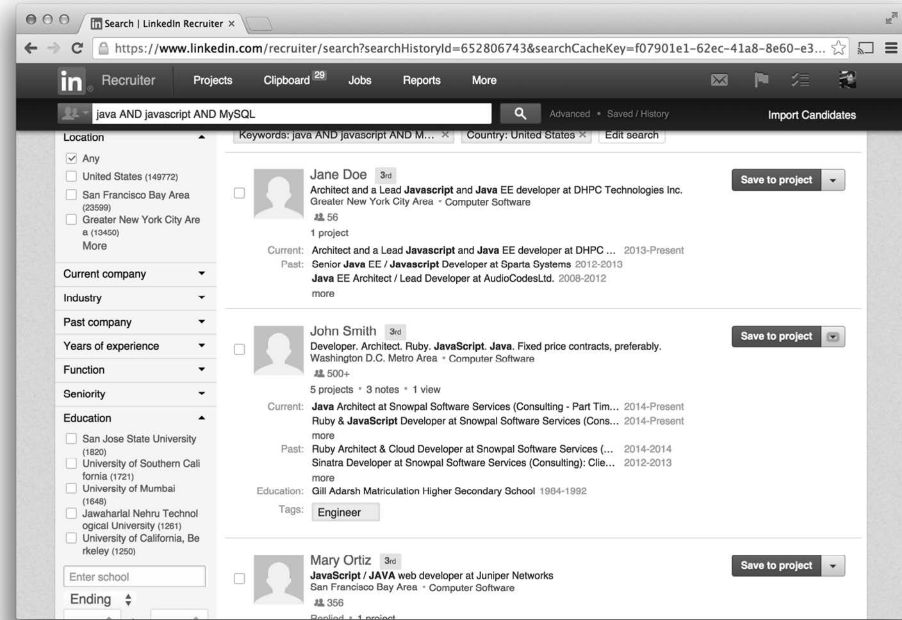


图 10-3：LinkedIn Recruiter 搜索

如果想让招聘人员找到你，必须尽可能将你的 LinkedIn 个人资料填写完整。个人资料中的信息应该和简历中的信息大致相同。你应当填写工作经历（工作职务和简要描述），教育（学位、研究领域、主要项目和课程），主要技能（语言、技术以及所擅长的能力）并列出所有发表的作品（博客、文章、论文、专利、图书、演讲）以及项目（业余项目、开源项目）。最重要的条目要放在简历靠前的位置并使用重点符号代替分段，因为招聘人员只会花 15 秒左右扫一眼个人资料或简历。另外，一定要把关注点放在具体的成就上而不是职责上。例如，如果你是程序员，不要列出“负责 X 产品”（这一点可以从你的职位头衔中清楚地看到）或者“在协同环境中能够有效沟通”（这是毫无意义凑字数的文字），而是要写上“使用 Z 技术，从头开始开发了 Y 功能”。更好的做法则是对你的成就进行量化，比如“使用 Z 技术，在 3 个月内从头设计并开发了 Y 功能，使产品的用户参与度提升了 50%”。

除了 LinkedIn 之外，其他一些需要创建网络身份的地方包括 GitHub、Stack Overflow、Twitter、SlideShare 和个人主页或博客。要检查你的网络身份，只要在 Google 上搜索你的名字就可以了[2]，看看出现的结果是什么。这些结果是否能向潜在雇主正面展示你的情况？你究竟能不能被找到？大部分招聘经理和招聘人员都会执行这种精确的搜索，如果他们不能在 Google 或 LinkedIn 上找到你，你就是不存在的。

注 2：最好使用无痕浏览模式，因为 Google 会根据已登录用户去定制搜索结果。

10.1.4 在线职位搜索

尽管我们应该把主要时间（大概 80%）花在建立人脉和个人品牌上，但是花点时间（大概 20%）在网上用老办法去搜索职位也是不错的方法。要牢牢记住，大多数创业公司的职位，特别是处于初期的创业公司，是不会出现在传统的职位版块上的。相反，我们应该关注一些创业加速器、风险投资公司的投资列表，以及众筹网站、编程竞赛和程序员网站。读者可访问 http://www.hello-startup.net/resources/jobs/ 网站全面了解创业公司的职位资源，找到你感兴趣的创业公司并直接和他们接触——当然，更好的做法是通过你的人脉去实现这一切。

10.2 通过面试

你也许是一名非常出色的开发人员，也很适合那家公司，但是除非你知道如何在面试过程中展示自己，否则也有可能无法获得它的职位。原因很简单：面试的过程是让人畏怯的。通常面试官会用一个小时去决定是否要在接下来的几年和你一起工作。可是在一个小时内，你能真正发现一个人的什么呢？你所能使用的招聘过程已经重复地在许多面试和面试官身上使用了，虽然其中有些实践方法更为出色（阅读 11.4 节了解更多信息），但没有完美的方法。而且所有面试都要求你——作为应聘者，展示出面试所需的技能，这些技能有时候已经超出了你平时每天进行的编程活动。

但既然面试是一种技能，我们就可以通过实践把它做得更好。你需要掌握的关键就是能够在白板上编程，能够把思考的过程说出来，认识自己，了解公司，熟悉那些简短的、重复出现的计算机基本问题。

10.2.1 在白板上编程

在白板上编写代码是一种讨人厌的面试方式（阅读 11.4.2 节了解更多信息），但是却为大多数公司所采用。面试并不是天生适合编程的环境：我们是在用手编写代码而不是用键盘输入，没有语法高亮、没有 IDE、没有编译器、不能剪切粘贴、没有 Google、没有 Stack Overflow，也没有任何惯用的其他工具。这很痛苦，但我们又不得不做。请加以练习！

10.2.2 把思考的过程说出来

你可能习惯了默默地在脑海里解决问题。不幸的是，如果你这样做，面试官无法知道你是如何得到解决方案的，也不知道如果你被难住，要怎么帮助你，或者如何评估你的表现。因此，要习惯在思考问题时说出来。另外，如果你陷入苦苦思索，也没什么好奇怪的。大多数面试问题就是专门设计让你要动脑筋思索一番的，所以如果你不能立即想出答案，也不要觉得恐慌，可以继续把思考的过程说出来，提出一些想法，尝试举几个例子，尽可能取得一些进展。把思考问题的过程说出来可能会让你感觉不自然，但却是一种在很多地方都有用的技能（例如可以用在结对编程和设计会议上），也是面试的基本要求之一。请加以练习！

10.2.3 了解自己

除了编程问题之外，大多数面试官还会问一些关于你自己的问题。

- 谈谈你自己。
- 你以前做过什么项目？
- 你为什么要找一份新的工作？
- 你为什么想来这里工作？
- 你理想的工作是什么样的？
- 你想在 5 年内做什么，10 年内呢？
- 你最大的优势是什么，最大的劣势呢？
- 你最大的成就是什么？
- 你曾经解决的最难处理的 bug 是什么？
- 我应该再问些别的什么问题吗？

有些问题听起来可能很俗气，但是却经常出现。请加以练习！

10.2.4 了解公司

虽然是公司在对你进行面试，但你也应该对它们进行“面试”。创业公司的职位是一种严肃的承诺——它会在接下来的几年轻易吞噬掉你一半的清醒时间——所以你要尽可能多地了解它们。在面试之前，要研究公司及其员工：他们是谁？他们在过去做了什么？他们在以后将走向何方？你也应该在面试过程中花点时间咨询一些你无法通过研究找到答案的问题。不要在面试完成之后还不了解以下问题的答案。

- 该角色的期望是什么？
- 该职位能够取得什么样的成功？
- 谁是我的经理？
- 我将从事什么项目？
- 技术栈是什么？
- 工作时间怎么样？他们花多少时间在编程上，又花多少时间来开会？
- 如何构建和发布代码？
- 公司的使命和价值是什么？
- 办公室怎么样？
- 在这里工作，你最喜欢和最不喜欢什么？

几乎每一名面试官都会给你咨询的机会。要当场想出一些问题并非总是那么容易，如果你什么都想不起来要问，看起来会像个糟糕的应聘者。一定要加以练习！

10.2.5 简短的、重复的计算机基础问题

由于时间限制，大多数面试问题都不会是大型、开放、真实的问题，而都是一些在限定范围内测试基本知识的小问题，通常来自于计算机科学课程的介绍，属于数据结构、算法和设计思想方面的。涵盖的主题也同样只是一小部分——大 O 符号、字符串处理、数组处理、遍历列表、遍历树、迭代、递归和动态编程——这些内容经常会在大多数面

试中出现，所以要备上一本《程序员面试金典》[3] 和《程序员面试攻略》。这两本书涵盖了 90% 以上在一般面试中会遇到的问题，练习这两本书中的问题，将会明显增加你的机会。

10.3 如何对工作机会进行评估和谈判

你找到了喜欢的公司，通过了面试，现在得到了工作机会。你应该接受它吗？你希望得到什么？如何进行谈判呢？

工作机会最重要的就是其背后的人。你准备好每周花 40 个小时以上的时间和他们在一起吗？你能否从他们身上学到东西？他们对于你的职业成长是否有帮助？和他们在一起有趣吗？他们是否投入足够的精力并具有足够的才能让公司取得成功？你是不是觉得仅从几个小时的面试就要得出这样的判断是很困难的？现在你知道面试官的感觉是怎么样的了吧？

除了凭借自己的直觉，你并没有多少选择。通常也确实是这样，如果这些人有什么让你感觉不好，或者有人让你觉得不舒服，事情只可能变得更糟。换句话说，如果你和他们有很好的对话，在他们身上发现了同样的激情，充满欢声笑语，这也许就是一种健康的氛围。对你的直接上司更是要特别地关注，尝试弄清楚他们的管理风格，他们的直接上级是怎么看待他们的，他们为你设定了什么样的角色和职业路径。这样的一个人对你的成功以及公司的愉悦氛围有着极大的影响。老话说得好：人们不是离开公司，而是离开经理。

如果公司的人都是人中龙凤，接下来要考虑的应该是业务本身。这是你所充满热情的产品吗？你想在接下来的几年在它上面花几千个小时吗？你自己会使用这样的产品吗？你会自豪地把这样的东西展示给父母看吗？你可能会从事自己并不感兴趣的无聊行业或者产品，但却难做出色，并感到快乐。

最后，如果你对公司的人和产品都感到满意，你就可以开始考虑这个职位经济方面的情况了。报酬有三种主要的形式：薪水、股权和福利。

10.3.1 薪水

考虑薪水时有两个门槛：

- 是否能够满足我的生活方式？
- 对于我所从事的工作是合理的水准吗？

第一个门槛取决于个人：刚从学校毕业、合租公寓房间的人与拥有房子、贷款、孩子的人的薪水需求是不同的。如果工作的薪水并不能满足基本的生活费用，你只能拒绝这份工作或者通过谈判要求得到更高的薪水。支付租金和吃饱饭可不是可有可无的。

如果这份工作跨过了第一个门槛，就应该看看你和竞争者相比情况如何。可以使用在线薪水计算器得到某区域类似职位的平均薪水。如果这份工作覆盖了生活支出，但是低于市场水平，那么只有在工作的其他部分——股权和福利方面有所弥补才是值得的。

注 3：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1010。——编者注

10.3.2 股权

什么是股权？什么是股票期权？什么是 IPO？对于在刚刚建立的创业公司工作的大多数程序员而言，股权才是工作机会中最重要的部分，而薪水不是。然而，刚从学校出来的时候，我并不知道股权是什么，也经常因为太尴尬而不知道怎么问。在网上搜索也得不到什么帮助，因为大多数搜索结果都使用了很多令人费解的法律和金融术语。这导致了我在职业生涯的早期对股权并没有多少关注，错失了很多潜在的收入。我采访过的许多程序员也都有类似的经历。

> 我在 Oracle 和 Facebook 之间犹豫。如果 Facebook 不是给我提供了比 Oracle 更高的薪水，我不能确定自己是否还会选择这个工作机会，这有点可笑，因为薪水在整个薪酬中仅仅占很小的百分比。但我不知道股权是什么，它对我来说什么都不是，那只是以后的事情。
>
> ——Daniel Kim，Facebook、Instagram 软件工程师

以下是对股权以及股权为什么重要的简单介绍。我希望自己在职业生涯中能够更早地了解这些东西，但愿这些知识对你有所帮助！

1. 什么是股票

假设朋友二人开了一家公司，每个人都想拥有公司的 50%，但如何才能把它记录下来呢？他们需要的只是某种方便的机制，能够记录对公司的所有权以及当他们将公司的一部分卖给投资者或员工的过程中，这些所有权是如何变化的。对于现代公司来说，这种机制就是**股票**。股票的每一股代表着一小部分的公司所有权。过去，公司会将股票信息打印在纸上，成为股权证明。但现在，大多数股票信息都是用数字化的方式记录的。

例如，两个创始人可以决定公司的所有权由 1000 万股来代表，每个创始人获得其中的 500 万股。股份的数量是任意的，创始人可以随意决定是用 1000 股还是 10 亿股去代表公司。然而，你需要知道股份的总数（通常称为**总发行股数**），才能知道你拥有公司的多少百分比。在我们的例子中，每一个创始人都拥有 1000 万的总发行股数中的 500 万股，也就是公司的 50%。

2. 员工如何获得股票

现在假设两个创始人要开始招聘，他们所提供给员工的激励之一就是员工拥有公司的一部分，通常这是通过发行新的股票并分配作为报酬来实现的。举个例子，创始人可能会建立 100 万股的池子，专门留作招聘用。这些股票是他们无中生有创造出来的（就像打印钱一样）。但是注意，现在总发行股数变成了 1100 万股，每个创始人仍然有 500 万股，但他们现在拥有公司的 45% 而不是 50%。这种所有权的减少被称为**稀释**。

现有的股东在公司每次发行新股票用于招聘或分配给投资者时，股权都会被稀释。这听起来好像对创始人不是什么好事，但是他们的赌注就是招聘到的新人或者获得的更多投资可以使公司更有价值，高于股权稀释带来的损失。例如，假设公司在联合创始人开始招聘之前价值 50 万美元，每一名联合创始人拥有 25 万美元。发行了 100 万股用于招聘之后，他们能够招聘到 5 名员工，这些员工将公司的价值提升到了 100 万美元。由于稀释的作用，每名联合创始人现在拥有公司的份额更少（是 45%，而不是 50%），但是更少的份额实际上却有更高的价值（价值 45 万美元，而不是 25 万美元）。换句话说，所有股

东分得的蛋糕比例变小并没有问题，只要这个蛋糕做得更大就可以了。

如果你获得了在创业公司工作的机会，也许会分配给你一定数量的股份——这就是你的**股权**。例如，你可能得到的工作会分给你 10 万股股票。这个数字本身没有什么意义，你需要知道公司发行的股份总共有多少。当然，如果这些股份是专门为你而留，那固然好；如果不是的话，就要问问你的股份在公司**完全稀释的股份**中占多少比例。这样公司就不仅要告诉你已经分配出的股份有多少，还要告诉你在未来应该分出多少股份以及专门留出来用于招聘的期权池有多大。如果公司拒绝告诉你这个数字，就是一个危险信号，你不应该加入其中。如果可以得到这个数字，你还应该问问未来的募资和招聘计划，这样就能对所占有的股份以后可能被稀释的程度有大概的了解。在前面举的例子中，我们知道公司的总发行股数是 1100 万股，所以你拥有的是公司的 10 万 /1100 万 = 0.9%。

3. 什么是股份兑现

你通常不会一开始就得到所有的股份，这些股份会在一段时期内分配给你，这就是所谓的股份**兑现**，通常兑现计划会持续 4 年：最开始没有任何股份，第一年过后会得到 25% 的股份（**底线**），剩余的 75% 会在接下来的 36 个月内平均分配。这种做法是为了保证你的股票能够持续增加，作为对你的一种激励，让你在公司留更长时间。若是没有兑现时间，你就可以在第二天离职并带走 10 万股股份。通常来说，公司中的每个人，包括联合创始人，都要服从于兑现计划。

4. 什么是股票期权

即便股份兑现，公司也不会将股票交到你手上。通常，实际得到兑现的只是**股票期权**，就是一种以固定价格（称为**履约价格**）购买（又称为**行使**）股票的权力。这一股票价格是指你得到工作机会时的**股票公平市价**（fair market value，FMV）。FMV 又是从何而来的呢？它通常是由董事会根据一种称为 409A 的估值方法计算出来的。

例如，如果 409A 对公司的估值是 110 万美元，履约价格就是 110 万美元 / 1100 万股 = 每股 0.1 美元。在公司工作 4 年之后，你可以兑现你的所有股票期权，你只有行使这些期权才能够处理它们。要行使你的所有股票期权，你必须要支付 10 万股 × 每股 0.1 美元 = 1 万美元。只有在行使了你的股份之后，你才能够实际把它们卖出去。当然，你的目标是以比支付价格高得多的价格把它们卖出去。

5. 如何卖出你的股票

开始的时候，股票份额是一文不值的，因为你不能把它卖给任何人（不能**流通**）。有两种常见的**流动性事件**（又称为**退出**），使股票变得有价值。

IPO：*首次公开募股*（Initial Public Offering）

　　是指公司将股票卖给公众。IPO 有助于公司募集资金，允许股东（联合创始人、员工和投资者）将他们的股票卖给市场。

收购

　　当公司被收购时，另一家公司用现金或他们自己的股票，从你的公司股东（联合创始人、员工和投资者）手中购买股票。

举个例子，如果你的公司进行了 IPO，股票的价格是每股 1 美元，你就能够以 10 万股 × 每股 1 美元 = 10 万美元的价格卖出你的股份。同样，如果你的公司被收购，收购者可能

同意以每股 1 美元的价格收购你的股票，这样的话你同样得到的是 10 万美元[4]。

6. 股票如何上税

我们从股票期权获得的所有收益都要上税吗？[5] 在上面 IPO 的例子中，你的收益是 10 万美元 – 1 万美元 = 9 万美元。这种收益纳税的有关法律是很复杂的，详细讨论已经超出了本书的范围。如果你曾经在股票方面做出过重大决定，一定要和税务专家一起处理。通常只需要支付几百美元，这点钱只占你要处理的资金的一小部分，或者远比你在以后去处理税务问题要便宜得多。为了帮助你掌握所要了解的东西，我会简单地讨论与股票期权有关的两个主要税务问题：短期和长期资本收益的对比以及行使期权的问题。

7. 短期和长期资本收益的对比

如果你在行使股票期权后少于一年零一天将其卖出，卖出这些期权所获得的收益就会作为**短期资本收益**被征税。也就是说，需要像正常收入一样纳税，好像是薪水的一部分。如果你每年赚 5 万美元，然后像上面所举的 IPO 的例子一样将股票全部卖出，那么你当年的应税收入就是 5 万美元 + 9 万美元 = 14 万美元。使用在线税收计算器可以得出，这一收入的联邦税率大概是 24%，也就是 33 600 美元。

如果你行使了这些股票，拿在手上至少一年零一天后再卖出，收益就会以**长期资本收益**被征税，这样会适用较低的税率。5 万美元薪水的联邦税大概是 12%，也就是 6000 美元，而 9 万美元股票的联邦税率是 15%，也就是 13 500 美元，总共是 19 500 美元。所以只要持有股票至少一年零一天，你在联邦税上就少了 14 100 美元。

8. 行使股票期权的问题

从上面的例子中，似乎可以明显看出尽可能早地行使股票期权会拥有巨大的税收优势。每当有股票兑现时，你可能禁不住想立即行使权利，让“一年零一天”的长期资本收益的时钟开始计时。事实上，有些公司甚至会让你**提前行使**股权，允许你在股权兑换之前行使。如果公司允许你提前行使，你就可以在加入公司的那天行使所有股权。4 年多之后，当你的所有股票都兑现时，你就可以把它们卖出去，立即获得长期资本收益的税率。

然而，有两个问题你需要知道。第一个问题很明显：行使股权要花钱。在上面的例子上，你需要花 10 万股 × 每股 0.1 美元 = 1 万美元。如果你加入公司的时间比较晚，FMV 已经上升，你的工作的履约价格也许会更高，比如每股 3 美元，所以行使期权将要花费 10 万股 × 每股 3 美元 = 30 万美元。并不是每个人手头都有这么多钱。即便有这些钱，它也是一种高风险的投资，因为你无法保证公司不会在几个月后破产，让所有的股票变得一文不值。

第二个要注意的地方更加微妙：当你行使股权的时候，如果股票的价格比你的履约价更高，你就必须支付履约价和市场价之间的差价税——即便你没有卖出任何股票。例如，假设你在 2014 年 1 月加入一家创业公司，当时的履约价是 0.1 美元。你不能确定它是否会进行 IPO，所以你没有行使这些股份。到了 2018 年 1 月，IPO 的可能性已经很大，所以你决定行使你的所有股份。问题就在这里：到 2018 年 1 月时，公司的公平市场价格可能已经上升了。我们假设现在是每股 5 美元，你的履约价格是固定在合同中的，所以你仍然可以支付每股 0.1 美元去行使你的股票期权，总共就是 10 万股 × 每股 0.1 美元 =

注 4：本章的所有计算都经过了简化。

注 5：每个国家的税法都不同，这一节描述的详细内容只适用于美国。

1 万美元。然而，你的股份现在价值 10 万股 × 每股 5 美元 = 50 万美元，所以你“赚”了 50 万美元 – 1 万美元 = 49 万美元。事实却是，你不能卖出这些股票，也知道这些收益跟你没有关系，却仍然需要为它上税。

这种收益的税率取决于股票的类型。**激励性股票期权**（incentive stock options，ISO）会根据可替代最小税率（alternative minimum tax rate）进行征税，而**非法定股票期权**（non-qualified stock options，NSO）则会按照短期资本收益来征税。我们假设你手中拿的是 ISO，你的 AMT 税率计算出来是 28%。这意味着你不但要支付 1 万美元去执行你的股份，还要支付 49 万美元 × 0.28 = 137 200 美元的税！而且你仍然无法保证在未来是否会出现流动性事件，或者说，即便出现了这样的事件，股票的价格还能保证超过每股 5 美元。

9. 进行研究

许多人都会被行使股份的开销及其相关税弄得焦头烂额甚至破产。这些基本知识只不过触及其中所涵盖的法律和税收内涵的皮毛，也不构成法律或经济上的建议。你需要和税务专业人士一起，彻底对股权进行研究，确保你能弄清楚它的所有情况。读者可以阅读 *An Introduction to Stock and Options* 进一步了解相关内容。

10. 我的股票期权价值多少

现在你明白了股票的基本知识，那么是不是多点股权少点薪水也无妨？这个问题可以从三个角度去考虑。第一个角度是把你的股权和所在地区类似的开发人员和公司进行比较。有一种方法可以做到这一点，就是使用 Wealthfront 的在线薪水和股权计算器。另一种方法是查阅 http://www.hello-startup.net/resources/equity/，上面有一张表格，根据工作角色、资历和员工数量，列出了在创业公司工作通常可以获得的股权数量。

第二个角度是把较低的薪水看作是对公司和你的职业的投资，看作是为了以后得到大回报的机会。这样的回报能有多大？谁都不知道。这其中有太多需要考虑的因素，所以你能做的只是进行大量的猜测和假设，得出很多种可能性。下面是一个评估风险和回报的公式（但是要牢牢记住，这只是一个简化的计算，不完全准确；另外，还要注意这个公式假设股票一兑现就被立即行使权利，而没有考虑纳税的因素）。

```
A = 工作薪水和公平市场薪水之间的差异
B = 你希望在该创业公司工作的年限
C = 行使股权的成本
D = 你拥有该公司的百分比数
E = 投资者的投入
F = 公司在成功退出时的价值

投入 = (A*B)+C
回报 = D*(F–E)
```

假设公司提供给你的薪水是每年 5 万美元，这一职位的公平市场薪水是 6.5 万美元。意味着你每年“投资”A = 6.5 万美元 – 5 万美元 = 1.5 万美元，用于换取未来从股权中可能获得的回报。你预期在公司工作 B = 4 年，这个年限通常足以兑现你所有的股票。提供给你的工作中，你以每股 0.1 美元的履约价格获得了 10 万股股票，所以要行使所有股份，必须花费 C = 10 万股 × 每股 0.1 美元 = 1 万美元。这意味着你在这家创业公司中的投入如下：

```
A = 6.5万美元 - 5万美元 = 1.5万美元
B = 4年
C = 10万股 × 每股0.1美元 = 1万美元

投入 = (1.5万美元 × 4) + 1万美元 = 7万美元
```

在你获得股票时，股票的总发行量是 110 万股，所以你行使的 10 万股代表了公司的 0.9%。然而到你达到退出条件时，你的股份可能已经被稀释了，因为公司会建立新的期权池，引入新的员工或投资者。我们假设 4 年之后你的股票被稀释了 50%，那么 10 万股最终代表了公司的 0.45%，也就是 D = 0.0045。

投资者对于你从股票中预期得到的回报有着巨大的影响。这是因为投资者通常得到的是**优先股**，意味着他们可以在任何人之前先退出而获利。例如，如果公司以前获得了 100 万美元的投资，后来要让 100 万美元退出，投资者可以把他们的钱拿回来，而别的人是拿不到的。这就是所谓的**优先权**，而回报甚至可能是最初投资的多倍。例如，如果公司以两倍的优先权筹集了 100 万美元，现在要让 300 万美元退出，投资者可以获得 200 万美元，而其他的所有人再对剩余的 100 万进行分割。投资者还可以应用其他的一些条款，比如**参股**，让他们在其他人之前从退出中获取更多的价值。要弄清楚所有的规则和招数是很复杂的，你需要读遍公司的所有条款说明才能精确地理解，所以通常能做到的也就是猜个大概。开始的时候，你可以对投资者做个大概的评估：

```
E = 募集资金 × 优先系数 × 经验参数
```

当你获得工作机会时，需要了解公司过去从投资者那里筹集了多少资金、在未来计划筹集多少资金，以及涉及的优先权（现在通常是 1 倍）。创业公司筹集大量资金并不总是很好的信号，因为这意味着为了让每个人都对回报满意，不得不安排更大规模的退出。其中的经验系数得由你来定，但必须保守一点。我通常用的是 3 倍的经验系数，因为投资者会用许多花招去增加他们的所得。

如果公司成功退出，公司将价值多少呢？没有办法得出确切的答案，所以你应该试试不同的算法，看看可能的结果是怎么样的。你应该将公司当前的估值和类似公司的估值做对比，以此为起点。例如，如果你在开发一个照片分享应用，也许就要把自己和 Flickr 做比较，该公司在 2005 年被雅虎以大约 2200 万美元的价格收购。关于 Flickr 募集多少资金现在没有数据可循，我们假设它募集了 100 万美元。

```
D = 0.0045
E = 100万美元 × 1倍的优先权系数 × 3倍的经验系数 = 300万美元
F = 2200万美元

投入 = (1.5万美元 × 4) + 1万美元 = 7万美元
回报 = 0.0045 × (2200万美元 - 300万美元) = 8.55万美元
```

现在看看如果你的公司更像 Photobucket，情况又如何？该公司获得了 1500 万美元的资助，并在 2007 年以 2.5 亿美元的价格被新闻集团收购：

```
D = 0.0045
E = 1500万美元 × 1倍的优先权系数 × 3倍的经验系数 = 4500万美元
F = 2.5亿美元

投入 = (1.5万美元 × 4) + 1万美元 = 7万美元
回报 = 0.0045 × (2.5亿美元 - 4500万美元) = 92.25万美元
```

最后，如果想用极端一点的例子，我们可以看看 Instagram，该公司募集了 5700 万美元的资金，并在 2012 年被 Facebook 以 10 亿美元收购：

```
D = 0.0045
E = 5700万美元 × 1倍优先权系数 × 3倍经验系数 = 1.71亿美元
F = 10亿美元

投入 = (1.5万美元 × 4) + 1万美元 = 7万美元
回报 = 0.0045 × (10亿美元 - 1.71亿美元) = 373.05万美元
```

作为参考，可以告诉大家风险投资支持的成功的创业公司的平均退出回报大概是 2.42 亿美元。

创业公司成功的概率又有多大呢？你可能听说过一个可怕的统计数据——90% 的创业都是失败的。当然，真正的数字取决于你如何定义“创业”和如何定义“失败”。举个例子，假设这是一家 VC 支持的公司，它的失败概率通常是 75%。如果你的公司像 Flickr 一样以 2200 万美元的价格收购，你就是投入 7 万美元以获得赚取 92.25 万美元的机会，或者以 1∶4 的概率去获取 13 倍的回报。这种情况看起来还不错。如果公司像 Instragram 那样以 10 亿美元退出，那么你就是投入 7 万美元以换取挣得 373 万美元的机会，或者以 1∶4 的概率去获得 53 倍的回报。现在来分析分析[6]。

那么，这是否值得呢？因人而异。对于某些人而言，失去 7 万美元的高可能性是一种不可接受的风险；对于其他一些人而言，如果能买来创业的经历、学习的机会、职业的成长、新的关系，以及获取巨大回报的机会，拿 7 万美元赌一把也是值得的。创业就是一种赌博，只有对风险有一定的容忍能力，你才应该加入其中。这也给我们带来了关于股权的第三个视角：它就是一张彩票，很可能没有回报；但如果给了你回报，它能够改变一切。

> 这实际上可以归结为个人的财务状况和目标。但在我看来，如果你是一个年轻人并位于硅谷，也许也可以下些赌注。多赚 1 万或 2 万美元的薪水并不能改变你的生活，赚 100 万以上才可以。
>
> 所以，如果你在这里，也许你也会玩这个游戏。掷几次骰子，在 3 家创业公司工作六七年，尽力去尝试、学习，然后稳定下来在 Google 工作，或者如果喜欢的话也可以继续在创业公司工作。你也许会命中，也许不会。如果没有命中，只能怪运气太差，也没有什么问题：你仍然得到薪水。也许你在创业公司得到的薪水会少一点，假设 3 年少了 5 万美元，又有多大关系呢？其实交了税之后，对存款也没什么影响。
>
> ——Nick Dellamaggiore，Linkedln、Coursera 软件工程师

11. 薪水与股权的对比

当提到创业公司的时候，许多技术人员都会告诉你为薪水进行谈判而不是为股权进行谈判是新手才会犯的错误。

注 6：要记住这种计算经过了很多简化，巨大的退出回报概率要比小的退出回报概率更低一些。Instagram 以 10 亿美元被收购是极其罕见的，所以这种概率很可能远低于 1∶4。

> 当我在 Google 面试的时候，大体上我根本就不知道自己在做什么。我的意思是，我就是个彻头彻尾的笨蛋。我记得我把所有时间都花在和面试官谈判薪水上，这完全是在争取错误的东西。差不多就是："好吧，我需要每年 5000 美元以上，而且我们正要从德国搬过来，所以要帮助我重新安家。"回想一下，其实在我到那儿之后很快就想到，这可以说是有史以来最愚蠢的谈判策略。我应该说"你们可以什么都不要付给我，只要给我股票就行了"。
>
> ——Kevin Scott，Linkedln 高级副总裁、Admob 副总裁、Google 主管

坦白讲，这有点儿马后炮了。如果在 Google 这样成功的公司工作，更多的股权当然是正确的选择。但大多数创业公司都是失败的，所以股权通常会一文不值。那么，我们仍然应该冒这样的风险吗？

> 我总是建议人们这样想问题：如果你打算为一家公司工作，你断定 1 万美元的薪水比起所获股票的上涨更值钱，那你就是在把宝贵的时间和精力压在错误的公司上——完全就是把赌注押在错误的地方。这是一种通常都可行的考虑公司的思维模式。
>
> ——Kevin Scott，Linkedln 高级副总裁、Admob 副总裁、Google 主管

创业并不全是和金钱相关的，但如果你不相信公司提供给你的股权会有什么价值，意味着你在怀疑公司究竟会不会成功。你最好干脆把这个工作机会拒绝掉。

10.3.3　福利

什么是福利？这些东西值多少钱？对于某些福利，你可以小算一番，评估它们的价值。例如，你可以评估一份健康保险计划值多少钱，看看它为这份工作增加了多少收入。如果公司每天提供免费的午餐，你也可以用每顿午餐的平均成本乘以你预期工作的平均天数，估算一下它的价值。

然而，许多福利的价值已经不能简单地用货币去进行计算了。你可以计算一顿免费的午餐价值多少，却无法计算你定期和同事吃午餐而建立起来的关系价值几何；你可以计算每一天假期都支付给你多少钱，但很难为你和朋友及家人一起做喜欢的事情标价；你可以计算报销参加兼职课程、听演讲或参加聚会小组的费用，但技能的提升以及工作中所受到的教育却是很难判断价值的。这些都是个人的决定，你必须通过自己的价值观来确定这些待遇价值多少。

10.3.4　谈判

获得了工作机会，你会为此而激动、快乐，松了一口气。现在要干什么呢？几乎任何情况下，你都应该在接受该工作之前进行谈判。在招聘市场，你是售卖自己技能的商人，而公司则是会尽可能支付你最低价格的客户。当然也不是说你得到的都是虚报低价的工作机会，而是说这一工作机会仅仅是这一讨价还价过程的第一步。几乎每一家公司都预料你会进行谈判，也几乎每一个职位在薪水、股权和福利方面都会有回旋的余地。

至于回旋的余地有多大就要取决于公司和职位了。提供给实习生和应届生的职位，特别是大公司提供的职位，通常都是标准化的。如果你刚刚进入这一行业，影响力很小，就

不要期望在薪水和签约奖金方面有多少提高的余地。如果你已经在该行业工作了好几年，通常会有 10%~20% 的浮动空间。如果你是首席开发人员或者高级总经理，30% 或更多的空间也是有可能的。注意，如果你希望在薪水上提高 10%，你就应该要求提高 15%。也许你会立马得到更大的数字，这当然很好，或者公司可能会向下还价为 10%，这就是你在开始时想要得到的水平。

最糟糕的情况通常是当你谈判的时候，公司就说他们无法再往上允诺了，也就是回到了一开始的工作条件。从另一方面看，只要你用专业的方式去处理这一问题，问一下也无妨，因为只是有潜在上升的可能。我几乎没有听说过仅仅因为谈判就丢失了工作机会，如果这种情况真的发生了，无论如何也不要在这样的环境中工作。

1. 应该告诉他们我的薪水吗

在面试的早期，即还远远未到确定录用的时候，许多招聘人员都会询问你之前的薪水。如果你告诉他们，他们就会把它当作录用你所需要付出的最少薪水，从而拉低了你的谈判能力。你应当礼貌地回绝，告诉招聘人员你更希望在正式讨论录用的时候再谈薪水的问题。

招聘人员可能会声称他们需要了解你之前的薪水，这样才不会因为薪水的问题浪费你的时间。在面试的早期确定你的薪水在一个合适的范围是没什么问题的，但这不需要暴露你自己的薪水。反之，你应该反过来问招聘人员，他们为这份工作提供的薪水范围是多少，告诉他们你会愉快地告知这是否在你可接受的范围内。如果招聘人员继续追问，就要提醒他们你的薪水属于私人秘密信息——可没有法律要求你泄漏。

2. 如果他们承诺后面再加薪呢

有些招聘人员会让你原样接受当前的工作条件，因为你会在以后得到大量的加薪和奖金。他们并不是完全在说谎，但是你应该假设所有没有明确写在合同内的东西都不会发生，承诺一年后有可能发奖金还不如提前给你加薪的保障。一般而言，你最可能在加入公司之前提高薪水。一旦你进入公司一两年，获得更高薪水或更多股份的难度就要大很多。

另外，你也应该考虑加薪和奖金是怎么计算的。如果你每年赚 5 万美元，那么 1 万美元的加薪就是一大笔。如果你每年赚 25 万美元，1 万美元的加薪并没有太大的影响。因此，大部分公司在加薪和奖金上都是按照你当前薪水的百分比来计算的。所以，让每年赚 5 万美元的人加薪 20% 就是 1 万美元，但是让每年赚 25 万美元的人加薪 20% 就是 5 万美元。提前谈判要求得到较高的薪水对你来说是一种巨大的优势，因为对以后所有加薪和奖金的提升都是有影响的。

3. 应该谈什么

薪水并不是面对工作机会唯一要谈的东西。你也可以为得到更多股权而进行谈判，特别是对创业公司来说，这点可能更有价值。如果该工作机会附带有股票期权，你可以要求获得签约奖金，以覆盖行使该股票期权的成本。如果你需要搬家，可以让公司为你支付重新安家的费用。如果你会在会议上做演讲，可以让公司清楚写明答应给你足够的时间进行这项工作，并支付差旅费。你也可以要求得到更多的假期、更高的 401k 缴存比例，或者其他对你比较重要的福利。

针对一个工作机会必须做出决定的时间也是可谈判的。大多数工作的“超期日期”都是假的，只是为了让你产生很紧迫的感觉。大多数软件公司都不会为固定的职位去招聘，

他们希望在一整年都有源源不断的人才流入。假设总是会有职位放开，假设他们已经投入数周或数月的时间来寻找你、对你进行面试并给你工作的机会，他们当然可以提供几个星期的时间让你考虑。如果你需要更多的时间，不妨告诉公司你不能匆忙做出这样一个重要的人生决定。

4. 如何谈判

大多数公司都不会根据你吸引人的个性和才智的估价来为你提供工作机会，他们会根据你的技能等级的平均水平给你提供工作机会。如果想获得更好的工作条件，你的目标就是让他们相信你比他们所想的水平更高。可以使用两种谈判策略。

第一种策略是把你能够摆上台面的技能、具备的经验拿出来讨论，并使用一些市场数据，让公司确信你不止值这么多。遗憾的是，除非你在行业中大名鼎鼎，否则这样的谈判战术是不会有多少效果的，因为它仅仅是你针对雇主观点的意见。这里你唯一的谈判优势就是说不，公司已经投入了许多时间和金钱去寻找你、联系你，对你进行电话面试，让你来到现场，对你进行面试和背景调查，决定雇用你并提供给你一个工作条件，所以你可以让大多数公司在薪水或奖金上给你一小点提升，仅仅因为他们不想让全部努力白白浪费。

第二种，也是更有效的策略，是通过获得竞争性的职位，向公司证明你不止值这么多。

5. 竞争性职位

作为职位的候选人，你最大的谈判影响力是竞争性的工作机会。另一家公司的工作机会是你在市场上价值多少的硬数据的象征，无须让你的意见和雇主的意见去一决高下。如果多家公司争相要你，你看起来就更有价值，你也许会让他们进入争夺大战中。我最喜欢的谈判策略之一就是像这样交谈：“嗨，A 公司，谢谢你提供了这个工作机会，我想在你们和 B 公司、C 公司之间做出选择，B 公司给我多提供了 X 的薪水，C 公司则给我多提供了 Y 的股权。我对你们的工作更感兴趣，也喜欢和你们在一起工作，但是却很难牺牲那么大的收入，你们能让这一决定对我变得更容易些吗？”

当你有多种选择的时候，你会发现使用一种最强大的手段会让谈判变得更加容易，那就是离开。这就是为什么我们总是应该同时面试多家公司。也是为什么面试工作的最好时机是你不需要工作的时候。如果你不顾一切地想要得到一份工作，就更可能接受任何正好能提供给你的工作。如果你已经有了可以接受的工作，当谈判达不到要求时，你也有路可退。所以要保持每隔几年看看招聘市场的习惯，即便你对当前的工作很满意，看看有什么机会出现或者确保自己当前的工作得到合理的对待，也是一种好的做法。

10.4 小结

在本章的开头，我让你写下了你希望找到的工作的性质。

- 什么行业以及什么产品类型？
- 什么技术？
- 什么商业模型？
- 什么职位角色？
- 什么待遇和福利？

现在，在本章的结尾，你应该能够反过来，把自己放在创业公司创始人的角色里，写下你想要招聘到的人员的品质。下面列出了几个你可能会写下来的例子。

- 聪明。
- 能把事情做好。
- 很好的文化契合。

最后，请再提供一个清单，说说作为创业公司创始人，如何对你希望找到的品质进行评估，例如针对上面提到的三个品质。

要评估某人是否聪明，你需要知道他们是如何执行现实中的任务的。如果你以前和他们一起工作过，或者他们是你信任的人引荐的，这是很简单的。如果不属于这两种情况，你只能在面试过程中去尽力找出答案，比如通过一些技术问题去实现。

要评估某人是否能够把事情做好，你需要了解他们在过去完成了哪些工作。如果你以前和他们一起工作过，或者他们是你信任的人引荐的，这是很简单的。如果不属于这两种情况，你应当尽力在网上搜索他们去找出答案。

要评估某人是否能够很好地适应文化，你需要知道他是否与你们有共同的价值。如果你以前和他们一起工作过，或者他们是你信任的人引荐的，这是很简单的。如果不属于这两种情况，只能通过面试尽力去找出答案，譬如可以问问应聘者过去都有些什么样的行动，或者问问他们未来的志向。

这种实践的目的是让你从雇主（阅读第 11 章了解更多信息）的角度看待招聘的过程。了解这些内容之后，你应该更清楚如何获得在创业公司工作的机会：你需要利用你的人脉、需要建立网上身份、需要练习面试，最为重要的是，需要了解你在公司文化、机会和工作条件方面需要什么。

第 11 章

招兵买马

11.1 创业与人密不可分

人是公司最重要的部分。选择正确的人比选择正确的产品、市场推广策略、技术栈或者编程方法论更加重要。这意味着招聘是你要做的最重要的事情，同时也是最难的一件事情。你需要找到与公司文化相契合的人，他要具备合适的技能，对你从事的工作感兴趣，并且在合适的时间点可供你使用，愿意接受创业公司的薪水。而且你还需要不断重复地去找符合这些条件的人。

对于创业公司招聘，最好的一条建议就是：别做这件事，或者至少先别做这件事。招聘更多的人意味着资金消耗更快，企业经营更复杂，决策更缓慢，要花更多的时间去搜索、面试和培训。最好的创业公司就是用较少的资源做较多的事。创业公司应当以小规模为荣，通过小团队去完成尽可能多的事。这能教你学会如何保持专注、更好地权衡取舍，并培养追求高影响力和高效率的热情。事实上，招聘更多的人并不一定意味着你可以完成更多的事情。9.2 节讨论过，沟通的开销会随团队规模平方增长，这也解释了为什么较大的公司不能像较小的公司那样快速转变。你应当尽可能保持小规模。

那怎么才能知道何时开始招聘呢？我们要问的关键问题不是“如果我们招了人可以做什么”，而是“我们不招人的话无法做什么”。如果对业务至关重要的事在没有新员工的情况下无法完成，不管你们多么有创造性，都是时候开始招聘了。本章将讨论要招聘什么人、如何找到出色的应聘者、面试的过程是怎么样的，以及如何提供一个别人无法拒绝的工作机会。

11.2 招聘什么人

要招聘出色的人，首先要了解什么样的人是你所寻找的。我们要考虑的招聘的主要类型有合伙人、早期员工、后期员工以及具有 10 倍能力的开发人员。

11.2.1 合伙人

在 Paul Graham 的"杀死创业公司的 18 个错误"清单中，排名第一的就是单一创始人。为什么呢？因为创办一家公司的艰难是一个人难以承受的，它会带来巨大的压力、风险和艰辛的工作，除非身边有其他人一起分担，否则大多数人是无法独自应对的。你需要其他人来弥补你的弱点，对你的想法给出反馈，在你犯了错的时候为你鼓劲。

> 创业的低谷非常之深，几乎没有人可以独自承受这一切。如果有多名创始人，团队精神会使他们以看似违背能量守恒定律的方式凝聚在一起。每个人都认为"我不能让我的朋友失望"，这是人性中最强大的力量之一，在仅有一名创始人的情况下，他将失去这一力量。
>
> ——Paul Graham，Y Combinator 联合创始人，
> 硅谷创业教父，《黑客与画家》作者

当投资者看到只有一个创始人时，他们不仅会推断这个创始人不可能独自获得成功，还会认为这个创始人无法说服他的朋友加入进来。这是一个不好的信号，也会使资金筹集变得困难。

因此，两个创始人通常是通往成功的最好途径。你可以平均分配工作和股权，也没有政治操作的空间。三个创始人也没有问题：只是在职责划分上会更困难一些，但总可以通过投票决定。一旦超过了三人，事情会变得越来越不稳定，做出决定也变得更加困难，职责和股权的分配也更难处理，更有可能出现政治问题和内耗。

但我们也不能轻率挑选创业伙伴，因为你要和他们相处很长时间——成功的创业都是以 10 年为量级的（阅读 1.4.1 节了解更多信息）。你应该和可能的创业伙伴已经建立起了较长的、经过考验的关系，才有可能和他们一起开创公司。创业从某种程度上就像去打仗，如果你和创业伙伴已经在过去经历了战争的洗礼，你就知道以后也可以相互依靠，这是最好不过的。

这就是为什么原来的同事通常会成为最佳创业伙伴：如果你们之前已经一起成功地做了些事情，就更有可能再创辉煌。如果你以前和大学同学一起做过项目、一起学习、一起通宵熬夜，这样的人也是不错的选择。朋友和家人则可能是下一个最好的选择，但是可能会牵扯更多的风险[1]。如果你之前从未和他们一起工作过，也许你并不清楚他们的技能如何，你们如何在压力下相处。此外，一旦个人关系不稳定，就更难以客观地做出好的商业决定。也可以这么说，你认识和信任的人几乎总是比完全陌生的人要好一些。如果你通过社交活动或在"创业伙伴之约"活动上聊了一个小时，就和遇到的创业伙伴去创立了一家公司，就像喝了点酒就要建立家庭或者在拉斯维加斯和陌生人奉子成婚——这并不是建立长期关系的最好基础。

我们应当在创业伙伴身上寻找四种品质。第一，要找"敏思笃行"的人。要寻找你认识的那些不管有什么障碍都能克服的人。第二，要寻找具有互补技能的人。例如，如果你是程序员并善于开发产品，就要寻找能够处理好销售和市场推广，并善于发现客户的创业伙伴。第三，创业伙伴通常在年龄、经济状况和动机上应该是类似的。如果一个创始人寄希望于赚快钱，而另一个则对改变世界有长期的愿景，这是很难成功的。第四，也是最为重要的，就是你要找到能够信任的人。要建立信任，你必须了解合伙人的经历，所以也再次说明了为什么同事、同学和朋友是最合适的创业伙伴。

注 1：研究表明，创始团队中每一段新的友谊会将犯错的概率提高将近 30%。

> 存在分歧是我们在尝试创业的过程中不可避免的。你不可能让三个创业伙伴总是一致点头。但如果你们彼此间出现了信任危机，那就成了分裂的基础，这和常见的分歧或大声争论是非常不同的。我想这可能是最大的风险，你可以修复bug，可以转变方向，可以改变几乎一切，但你无法解决人的问题。如果创业伙伴相互间不信任，那你唯有把工作叫停，或者你们中有人将不得不离开。
>
> ——Vikram Rangnekar，Voiceroute、Socialwok 联合创始人

只有一个创始人也许是“杀死创业公司的 18 个错误”中排名第一的错误，但是“创始人之间的斗争”也在清单上。想要让所有创始人感觉平等，确保他们所有人都在一条船上，通常就应该平均分割股权（阅读 10.3.1 节了解更多信息）并实施兑现计划（阅读 10.3.2 节了解更多信息）。不管公司创立之前是谁贡献的点子，也不管他们已经做了多少工作，平均的股权分割计划都是最佳的选择[2]。平均算下来，打造成功的创业公司要花 10 年的时间，所以与其在第一天就对股权百分比争论不休并产生矛盾，还不如确保每一个人都有相同的激励，以长远的眼光去克服各种困难。

11.2.2　早期员工

创业时要花些时间去寻找最开始的几个员工。做了错误的产品或使用了错误的技术，我们都可以从头再来，但如果没有招对最开始的 5~10 个员工，却可能将公司置于死地。早期的员工和联合创始人一起，对公司文化的定义负有责任，出错的后果是无法承担的。例如，Airbnb 的创始人花了将近 6 个月的时间，面试了数以百计的应聘者，才找到了他们最初的员工。他们的 CEO Brian Chesky 做出了这样的解释。

> 最初的技术人员就像把 DNA 引入你的公司。如果成功了，公司中将会有 1000 个像他那样的人。这不是让某个人去实现我们接下来要提供给用户的 3 个功能，而是一件更长期也更持久的事，也就是我是否要和 100 个或 1000 个这样的人一起工作？
>
> ——Brian Chesky，Airbnb 联合创始人、CEO

Chesky 还问了应聘者“如果你剩下一年的生命，还会选择这份工作吗”，而且他只会录用回答“是”的应聘者。这是一个极端的例子，你可能不应该完全照搬（即便是 Chesky，后来也把问题改成了十年），但是这个问题真正的意义才是我们要放在心上的：尽你所能，确保自己找到正确的早期员工。

值得记住的是，在创业早期，每一个人都必须要做所有的事情。产品、需求、团队和技术全都在以非常急速的频率发生变化。某天你会对数据库查询进行性能调整；另一天也许又在调试 JavaScript 错误；在接下来的这周，你在想办法解决如何备份源代码；几天之后，你又在给投资者的融资演讲稿做收入预测。这意味着创业公司早期的技术人员应该主要是**多面手**或者**全栈工程师**。也就是说，这些人应该对很多方面的任务都很擅长，而不是某项单个任务的专家。

注 2：Y Combinator 最好的公司中，没有一家公司在股权分割上存在显著的不均等。

> 开始的时候，我只会聘请全栈的人，因为不想有人说："这不是我的工作。"在公司最开始的阶段，每个人头上都要顶着好多帽子。只有到了稍后期，我才会去招聘专才。
>
> ——Julia Grace，Weddinglovely 联合创始人、Tindie CTO

这里有一个重要的提醒：专才的反面并非是通才。不是任何方面的专家并不意味着你就擅长各个方面。真正的通才是那些多面手的人，你把一些新东西放在他们面前，他们就会把它带走并弄清楚它是如何工作的。他们拥有永不满足的好奇心，愿意对所有东西都稍做尝试。但你要是看看他们的简历，就会看到大量行业的经历（例如旅游、医药、社交网络、消费者、部署自动化、编程语言设计）以及职业角色（例如开发人员、技术主管、产品经理、设计师）。

也许对一个通才来说，最明显的特征并不是他已经具备的经验，而是他乐于去体验所有新的经历。如果你听到一个开发人员说"噢，不，我不是搞前端的"，或者因下一个项目不得不学习新的编程语言时有所畏缩，这样的人可能就不是通才。真正的通才喜欢学习新东西，不管问题的领域是什么，也不管他们是否已有相关经验。

这就是为什么有些人可能一从学校出来（甚至从学校辍学），就可以打造出世界上最成功的创业公司，他们可没有任何的经验（阅读 11.3.5 节了解更多信息）。正如马克·吐温所说："他们并不知道那是不可能的，所以他们做到了。"在创业招聘时，不要一遇到刚毕业的大学生或者其他没有经验的工程师就避而不见，只要他们对创业公司将抛给他们的各种东西表示出了学习热情，就应该加以考虑。对创业而言，态度胜过才能。

11.2.3 后期员工

随着公司的发展，问题会变得更加复杂：我们也需要找到能够扩大公司规模的方法，以应对更多用户、更多客户、更多流量和更多员工——这是专才派上用场的地方。当你需要调优数据库查询、建立复制或共享表格，也许就是时候去招聘一个专职 DBA 了；当你需要管理数以千计的服务器、把代码部署到多个数据中心、建立 24×7 的网站监控，也许就是时候去招募一个专职发布工程师了；当你的网站规模大到经常遭受黑客和垃圾邮件的攻击，也许就是时候招募一支专职安全团队了。但要提防招聘那些只擅长一种技能而别无他长的人。相反，我们需要 T 型的人，这一点已经在 2.1.1 节讨论过。

11.2.4 10倍能力的开发人员

一系列研究表明，开发人员之间的生产效率存在巨大的差异。举例来说，该领域最早的一项研究发现，"最好和最差的程序员初始编程时间之比大概是 20∶1；调试时间之比大概是 25∶1，程序规模之比是 5∶1，而程序执行速度之比是 10∶1"。最初的这项研究存在一些缺陷，但是之后的许多研究也发现了程序员能力上的一些类似差异——大概相差一个数量级，或者说是 10 倍。尽管在这样的研究中，**10 倍能力的开发人员**这一概念受到了相当多的嘲讽，许多人都怀疑这样的"摇滚明星式开发人员"是否可能存在，但是似乎不会有人怀疑明星般的运动员、艺术家、作家，当然，还有摇滚明星的存在。

问题是当人们提到"10 倍能力的开发人员"时，他们想到的是一个神秘的程序员，他可以在普通开发人员每编写一行代码的时候，就写出 10 行代码来。这种推论的缺陷在于，

编程效率和输入速度以及代码行数并没有什么关系。编程是一种创造性的职业，对同一个问题会有许多种解决方式。举个例子，我们可以想一想在开发一个软件产品（比如一个网站）的时候，汇聚了多少决定和创造性。你使用了什么语言？什么 Web 框架？如何存储数据？使用的是什么缓存技术？你把网站托管在哪里？如何监控网站？如何上传新的修改？如何存储代码？建立了什么类型的自动化测试？ 10 个普通程序员可能会在每一个步骤都做出质量“平均”的决定，这些决定的代价或好处是会成倍增加的。假设流量以指数式增长，这个普通的团队做出了一个普通的网站，其数据库是很难扩展的，主机也没有足够的冗余，版本控制也没有适当的备份，亦不存在监控。当这 10 个编码人员把全部时间都花在到处灭火上，他们能够有多少生产力呢？

如果一个程序员解决这个问题所花的工作量要少一个数量级，那么这样的一个程序员可能胜过一个团队。通过做出更好的决定和提出更有创造性的解决方案，10 倍能力的程序员也许可以避免埋头苦干好几个月。换句话说，我们并不是要编写**更多的**代码，而是要编写**正确**的代码。10 倍能力的程序员实际上是 10 倍能力的决策制定者。

这种情况并非只在编程领域出现。比方说，你是想要 10 个平均水平的科学家还是 1 个牛顿呢？ 10 个普通的科学家想不出运动定律、引力理论或者微积分原理，但 1 个牛顿却可以做到。你是想让埃隆 • 马斯克去运营公司，还是想把钥匙交给 10 个普通的执行官？ 10 个普通的执行官想不出 Paypal、电动汽车、可回收火箭和超级高铁，但 1 个埃隆 • 马斯克却可以做到。

这一切表明，超级明星程序员，就像超级明星运动员，是格外稀少的。如果你尝试根据只招聘“摇滚明星”的原则去制订自己的招聘策略，或者更糟的情况是，如果你的公司招摇过市，好像已经是一群“摇滚明星”，你就永远无法发展你的团队。开发人员在能力上也许会有巨大的差异，但是这一点不能掩盖下面这三个关键的事实：

(1) 开发人员的表现会根据不同的任务而有所不同；
(2) 开发人员可以逐步变得更好；
(3) 大多数软件都是由团队而不是由个人开发的。

第一个事实意味一个公司中 10 倍能力的开发人员在另一家公司中可能只是 1 倍能力甚至 0.1 倍能力的开发人员。文化、激情和使命才是至关重要的（阅读第 9 章了解更多信息）。第二个事实意味着建立成功创业公司的最好方法就是招聘到出色的开发人员，并为他们提供一个环境，使他们可以变得更好。有些人知道了 10 倍能力的开发人员的概念会觉得沮丧，因为他们认为自己永远不会做得那么出色，似乎伟大的开发人员是天生的，而不是后天的。但是在阅读了第 12 章之后，你就会发现许多证据表明，将精英和一般人区分开来的并不是天赋。而是实践练习。

这就是为什么我发现 10 倍能力的开发人员的概念后大受鼓舞，而不是沮丧。因为这表明程序员并不全都是一个模子刻出来、像机器人一样可以被呼来唤去的苦力。它意味着编程就像一门手艺，你可以把自己从学徒培养成工匠，并一直往上，成为大师级的手艺人。这也意味着，要达到大师级水平，靠的就是实践练习。运动精英每天要练习好几个小时才能提高自己的能力，如果你渴望成为精英级的程序员，也应该这么做。我们不妨试着将 10 倍能力的开发人员看作激励自己写更多代码的行为榜样，就像一个小孩可能会把乔丹看作是激励自己打篮球的行为榜样。同样，每一个专业运动团队都有一种“农场体系”

（例如高中、大学、小联盟、人才探子）去培养天才。如果你渴望建立一家精英软件公司，也应该这么做。举办技术访谈、报销课程和会议费用、鼓励撰写博客和开发开源软件、组织黑客马拉松、建立导师制，我们要成为能够培养 10 倍能力开发人员的公司，而不是尝试成为能招聘这样的人的公司。

最后，第三个因素也可以说是一个提醒——真正重要的其实不是个人的表现，而是团队的表现。和个人一样，团队的生产效率也是各不相同的，差不多也会有一个数量级的差异，部分原因是“出色的程序员容易聚集在某些组织中，而糟糕的程序员则会在其他组织”，部分原因则是“组织的因素，比如出色的组织能够定义出清晰的产品愿景、定义清晰的需求、协调团队成员间的工作，等等”。换句话说，我们不要关注是否能招聘到一个超级明星，而是要关注如何形成起一种可持续的招聘策略，如何形成一种组织结构，让每一个人都能高效地在一起工作（阅读 9.3 节了解更多信息）。

11.2.5　寻找什么

我们要在所有应聘者身上寻找的重要品质就是，他们应该是聪明并能把事情做好、能够很好地适应文化、有出色的沟通技能、能够友好相处的。

1. 聪明并能把事情做好

编程，特别是在创业公司中，所需要的聪明才智已经超过了知识本身。因为我们在创业中面临的挑战将会频繁地改变，所以有足够的聪明才智能适应新的挑战，比了解之前挑战的每一个微小细节更加重要。但是只有聪明才智是不够的，也许你和这样的聪明程序员共事过，他可以滔滔不绝地说上几个小时，介绍一种技术与另一种技术相比，在理论上有什么好处，他会花许多时间去撰写设计文档、架构图和 UML 图，但是从来没有实际给出过任何代码。我们有时候称他们为设计师，有时候又称他们为教授。不管怎么样，创业公司需要的不仅仅是聪明的人，还需要能够把事情做好的人。我们需要的是可以权衡取舍、可以利用不完美的信息做决定、能够理解完成比完美更好的人。

注意，我们要寻找的这种“聪明”和 IQ 或者漂亮的学历并没有多大关系，更多的是要有清晰的思维和解决问题的能力。你可以问应聘者一些你已经深入了解的问题，对他的这种能力有个大概的了解。你也可以讨论一些编程语言（比如深入地比较函数式编程和面向对象编程）、库和框架（例如让应聘者比较 Ruby on Rails 和 SpringMVC）、系统架构（例如如何为 Twitter 实现一个 URL 的缩短程序）、特定的问题域（例如给我讲讲实现推荐电影的机器学习算法），或者其他任何你可以步步深入发问的问题。你的目标并不是要了解应聘者现有的知识，而是要让他们进入提前并不知道答案的领域。你所寻找的这类聪明人能够将未知转变为已知，不管他们面临什么样的挑战，他们都能自如地应对不确定之事，他们会问正确的问题，他们可以快速地学习和适应。

我们可以使用类似的问题去找出能够“把事情做好”的人，但是不同的是，我们要关注应聘者深入了解的领域。在他们的简历上找出他们所感兴趣的东西，或者问他们一些与自己有关的最自豪的成就，一步步地深入发问，直到进入**你**事先不知道答案的领域。如果应聘者不能向你解释清楚，要么他们并非该项目的主要贡献者（所以实际上并没有“把事情做好”），要么就是没有足够的理解（所以并不“聪明”）。

> 对于背景，我想了解的是一个人完成了什么，而不是他参与了什么、不是他是其中的一员、不是他是见证者，或者只是在事情发生时在周围徘徊。
>
> 我要了解的是你所完成的，无论是在工作中还是在工作之外（通常更好）。可以是你在高中时开始做的事情，也可以是你在大学时创办的非盈利组织。如果你是程序员，可以是你做出了主要贡献的开源项目，诸如此类。
>
> 如果你什么都发现不了——如果应聘者一直都在循规蹈矩地做事情，展示的是他们上课上得好，考试考得好，找到了好的工作机会，相对于他们的起点来说，没有实现任何独特和引人注目的东西——他们可能并不是受自己内心的驱动去做事情，而你是无法改变他们的。
>
> ——Marc Andreessen，Netscape、LoudCloud、Opsware 和 Ning 联合创始人

2. 文化契合

聪明和把事情做好是必不可少的，但却并不够。你还要花大量时间和你所招聘的人在一起，在小型创业公司中更是如此。设想你要每天都和这个人一起吃午饭，要他们去评审你的代码，要花时间去了解他们或者从他们身上学习东西，当网站出了问题他们会在凌晨 3 点打电话给你。虽然你没必要希望招聘到能够变成朋友的人，但你肯定要避免招聘到你无法相处的人。在许多公司中，这种品格有一个好听的名字：**无混蛋规则**。这是一条很好的规则，但不是“混蛋”还不够，你真正需要的是找到很好地与你的公司**文化相契合**的人（阅读第 9 章，了解有关创业文化的深入讨论）。

我们要注意，不要把文化契合和个人取向混淆起来。一家公司的文化是其行动的方式，而文化契合就是找到能够像你一样去行动的人，因为他们拥有相同的核心价值（阅读 9.2.2 节了解更多信息）。而不是说要找到因为具有相同的背景，所以看起来像你或者和你有共同爱好的人。许多公司在这一点上都做错了。例如，下面引用了旧金山一家创业公司的博客，我把名字隐去了。

> 他穿着无可挑剔的西服……我偷瞥了一眼，发现当他走进来时，团队中的好几个人都在抬头看着他。我可以感觉到他们的失望之情。并不是说我们的着装要求很小气或很严格，所以我们会因为他没有遵循这一不成文的规定而取消他的录用资格，而是我们凭经验认为，穿着西服进来的人很少能够融入我们的团队。他就是没有通过这种“一起出去喝一杯”的测试，他甚至没有觉察到这一点。
>
> ——旧金山一家创业公司的创始人

喜欢啤酒并不是一种核心价值，厌恶西服也不是一种核心价值。认为着装要求无关紧要也许是一种核心价值，但是很明显，这家公司并没有这么做，所以这并不是他们的文化。相反，他们是根据喝酒和衣着的取向做出了招聘决定，这并不是文化契合的问题，而是偏见和歧视。

判断文化契合的正确方式就是寻找这样的应聘者，他们的行动所表现出来的核心价值与你们的核心价值是相吻合的。例如，Zappos 的核心价值之一就是“用服务震撼用户”。那么，他们是如何确保招聘到的每个人都对客户服务充满热情的呢，我们不妨看看。

> 每一名应聘进入我们总部的员工，不管他在什么部门或是什么头衔，都要通过一种培训，和我们忠诚客户团队（呼叫中心）的客户代表们所介绍的培训是相同的。不管是会计，还是律师，抑或软件开发人员——你都要经历完全相同的培训程序。
>
> 这是一个四周的培训程序，在培训中，我们会介绍公司的历史、客户服务的重要性、公司的长期愿景以及我们对公司文化的理念——之后你会实际接听两个星期的电话，接听客户来电。这实际上是再一次回到我们的信念——客户服务不应该仅仅是一个部门的事，它应该是整个公司的事。
>
> 在培训第一周结束时，我们为整个班级提供一个机会，对想退出的人提供 2000 美元（此外还支付他们已经工作的时间），这种机会每个星期都有，直到培训第四周结束为止。我们要确保留下来的员工不仅仅是为了一份薪水，我们希望员工能够信奉我们的长期愿景并愿意成为我们文化的一部分。最终，平均会有不到 1% 的人最后会择机退出。
>
> ——Tony Hsieh，《回头客战略》

这就是一种很好的文化过滤。如果你对客户服务没有热情，就不应该申请在 Zappos 工作。在提到依据员工的热情进行招聘时，也存在一个问题：公司之外的大多数人并不了解有关公司或其使命的任何东西，特别是当你还只是一个微小的创业公司时。如果你正在从事一些革命性的事情，他们甚至会认为你是疯狂的。因此，虽然过滤掉不能认同你价值的应聘者是没什么问题的，但你不能仅仅因为他们没有**提前具备**一种热情或者对你们的使命缺乏关注就把应聘者过滤掉。

注意这里“提前具备”是重点。我们寻找那些在之前工作过的公司充满热情的应聘者，或者对他们之前做的产品、使用过的技术或者对你所从事的行业充满热情的应聘者是没有问题的，但让他们对你的公司和使命充满热情就是你的任务了。你必须让使命清晰而可见：它应该在产品中明显地体现出来，在各种市场推广材料中都可以见到，出现在你的公司网页和每一个职位说明中。每个面试官都应该了解和讨论公司的使命，你也应该确保每个应聘者都能理解它，只有这样才能做出你的主观判断。

> 很多人都喜欢把热情一直挂在嘴边，但是如果你从来没有在公司工作过，怎么可能对公司充满热情呢？我觉得这是真正让人匪夷所思的，除非那是你自己的公司。另外，除非你自己就身处其中，否则也很难对某些东西有超级的热情。
>
> 我想一个人应该对自己有热情，他们要成为更好的自己并想提升自己的生活质量。如果我想让他们对 LinkedIn 充满热情，我知道如何让他们做到。但他们必须对自己的生活充满热情并愿意为之贡献，否则这样的热情并不会持久。
>
> ——Florina Xhabija Grosskurth，LinkedIn Web 开发人员、经理，Wealthfront 人事主管

3. 出色的沟通技能

沟通技能比技术上的技能更为重要。事实上，大多数技术上的技能都只不过是沟通技能的伪装。例如，编写整洁的代码主要还是以一种其他程序员可以理解的方式去编写代码（阅读第 6 章了解更多信息），即把你的意图清晰传递出来。因此，在同等条件下，人们

总是更希望拥有沟通技能更好的程序员。

当你考虑一个应聘者的时候，你是否能够和他愉快地交谈呢？如果公司中的某个子系统是他们负责的，他们能不能向新来的员工解释清楚系统是如何运转的？他们有没有维护博客？如果有的话，写出来的东西你能不能看得懂？他们能不能在会议上演讲？你能不能理解他们的代码？如果你们的角色调换过来，那名应聘者在面试你，他们会是公司的出色代表吗？

4. 我会乐于向他们报告

最后一项是对前面所有各项快速、理智的检查。如果你永远不想让这个人当你的老板，通常意味着你认为他们不够聪明，或者他们不知道如何把事情做好，或者他们不适应你的公司文化，又或者你无法理解他们。即便只是资历尚浅的应聘者，我们也不妨想想看，如果他们某一天成了主管，情况会怎么样。如果那样的想法让你感觉不舒服，你也许就不该把他们招聘进来。

11.3　寻找出色的人选

现在我们了解了要找什么样的开发人员，接下来再谈谈如何找到这样的人。大部分创业公司都低估了找到出色的人才所要花的时间。假设你是一个有两名合伙人的团队，想要招聘 12 名技术人员。你认为完成这一任务要花多少时间？答案是：大约 990 小时，或者在一整年里每周花大概 20 小时去招聘。

那我们要如何提升这一速度呢？为你的创业公司找到新人的最好方法是什么？通过招聘人员，还是通过代理？虽然可能都挺有用，但最好的起始点是推荐。

11.3.1　推荐

对公司来说，最好的招聘策略可以总结为一句话：每个人都是招聘人员。如果你要找到最佳候选人，可以先从你已经了解的人入手。创业伙伴们应该招聘他们已经了解的最出色的人；这些新招聘进来的人，又有他们自己的人脉，他们也应该努力去招聘他们了解的最出色的人，以此类推。推荐可以获得质量最高的员工，帮助你更快地填补职位空缺（推荐要花 29 天，而招聘网站则要花 45 天），而且留存率也是最高的（通过推荐进来的员工在两年后仍有 45% 的留存率，而通过求职平台进来的仅有 20%）。

推荐不是自然而然就会发生的，所以你必须主动。例如，花点时间亲自看看你在 LinkedIn、Facebook、Twitter 和手机上的联系人，联系所有可能有理由匹配的人。同样，当你参加会议、聚会小组、和投资者聊天，甚至和朋友或家人一起休闲时，不要害怕谈论你的公司和使命（阅读 9.2.1 节了解更多信息）。不要觉得讨厌或者那是爱出风头的表现，你要认识到，大多数创始人都花了 25%~50% 的时间坐在咖啡店里，向朋友谈论他们的愿景。此外，我们也要提供激励，让公司的每一个人都加入到推荐中来，比如现金、有趣的奖项或者对每一个通过推荐进来的员工给予认可。在早期，几乎所有员工都应该是通过推荐进来的。随着时间的推移，这个百分比会下降，但要努力永远不要让它掉到 40%~50% 以下。如果出现了这样的情况，表明你要么在鼓励推荐上做得还不够，要么，更糟糕的是，你的员工无法将你的公司推荐给他们的朋友。

11.3.2　雇主品牌化

通过推荐找到出色的应聘者有一种最好的方法，就是让出色的应聘者可以找到你。亲自去寻找应聘者并说服他们公司值得申请，需要大量辛苦的工作。但如果你能建立起公司形象，让应聘者想要在你的公司工作，你就在比赛中处于领先位置了。为此，你需要建立起出色雇主的**品牌**（阅读 4.2.4 节了解更多信息）。

其中的关键点在于：74% 的劳动者都是被动的应聘者。机会对他们是开放的，但他们并不会主动寻找工作，所以他们永远不会花时间去查看工作公告或者访问你的招聘网站。因此，建立雇主品牌去吸引技术人员的最好方式就是打造一些有价值的内容，让他们会因为自身的原因去查看，而不是因为他们正在寻找一份工作（阅读 4.2.2 节了解更多信息）。你需要的是这样的内容：当这些内容出现在 Hacker News、Reddit、Twitter 订阅或者 Google 搜索的结果中时，你会去点击它。例如，你可以建立一个技术博客，描述你的公司正在解决的问题类型，以及解决这些问题所采用的技术；你可以在会议上讨论这些技术，可以在 GitHub 上把这些技术开源；你也可以举办黑客马拉松、聚会小组和 Drinkup 活动，建立公司的 Twitter 账号、Facebook 网页、LinkedIn 公司网页和公司的主页，为你的技术和产品培养一个社区。

如果公司中的每个人都有一个强大的个人品牌，会比单独的公司品牌还要好。所以，我们要鼓励员工开通个人博客、个人 Twitter 账号、在 GitHub 上开通兼职项目、做演讲、写任意主题的论文，哪怕和公司没有什么关系。这样不仅会使他们成为更出色、更快乐的员工（将在第 12 章讨论），而且如果你的员工有强大的吸引力，他们也会吸引人们加入到你的公司中。

> 在我的个人经历中，最让我印象深刻的一件事就是撰写 Quora 答案。我写过一些答案，回答这样的一些问题：在 Pinterest 工作感觉怎么样，在 Pinterest 做一名前端工程师感觉怎么样，如何在 Google、Facebook 和 Pinterest 之间选择，等等。结果表明，那些正在考虑申请工作或者加入某个公司工作的人，喜欢在网上做做研究，很多人都会在无意中发现我写的东西。在 Pinterest 有不少新员工已经告诉我，他们加入进来就是因为喜欢在我的 Quora 文章中读到的东西。
>
> ——Tracy Chou，Quora、Pinterest 软件工程师

注意，你所制作的大部分内容不应该明显和招聘人员有关。例如，不要用招聘视频去打扰别人，大家应该知道我指的是什么：在一个快速平移的镜头下，出现了一个站在白板旁边或者打着乒乓球的人；在一首快节奏的背景轻音乐的伴奏下，出现了一个面试的场景，一名行政人员坐在中央，看着摄像头一侧的人，告诉他们所做的工作是多么有意义，接下来就是渐渐出现的公司 logo。实际上，没有公司之外的任何人会喜欢看这种片子。

相反，我们应该创建一些本身就有价值的内容—— 一些你的受众可以从中学习的内容和他们可以使用的工具。如果一个应聘者已经使用了你公司的开源软件、从你们的博客和技术演讲中学习到了新的技能、在网站关注着你的一些员工，他就更容易被招聘进你的公司。这样的应聘者也更有可能很好地适应你的公司。对你们的代码库提交了 pull 请求，或者在你们的技术演讲上咨询了问题的开发人员肯定对你们所使用的同一技术也充满热情，也许他们也会对在你们公司工作充满兴趣。

> 在 Typesafe，我们已经被宠坏了。因为大多数公司都是基于开源软件而建立的，我们一直能够通过开源社区去招聘人员——本质上这可以说是你能够想到的最长、最彻底的面试过程。我为 Akka 团队招聘的人多年来一直都在为我们的项目提交东西，我们都已经认识并能够舒服地在一起工作。
>
> 开源对我们是很有帮助的，因为如果人们已经在使用开源软件并对其充满热情，我们就可以先看看他们的 GitHub 代码库。这是再容易不过的了，你可以立即过滤出有价值的人，但也许还不是你所寻找的类型。
>
> ——Jonas Bonér，Triental Ab、Typesafe 联合创始人

11.3.3 在线搜索

即便你已经建立起了很好的推荐程序和网上的品牌，也仍然要亲自去网上搜索应聘者。http://www.hello-startup.net/resources/jobs 提供了我们可以使用的网站完整列表，其中包括创业加速器的网站集、风险投资公司网站集、众筹网站、程序员网站、编程竞赛，最后还有求职榜。我把求职榜放到了最后，是因为它们通常都是寻找应聘者最没有效率的渠道。如果你只是在求职榜上发一个职位（贴上去后就听天由命），你只能接触到主动的职位搜寻者，最好的情况也只是 1/4 的应聘者。对于程序员来说，这个比例甚至更低。我所知的最出色的程序员在他们的整个生涯中只会申请一个职位，就是大学毕业的时候申请的。此后，他们几乎毫无例外都是通过推荐或者因为有人去接触他们而获得职位。对于大多数程序员来说，招聘公告已经是淘汰的东西了。

事实上，没有人喜欢招聘公告，看看下面这个例子。

> 我们正在寻找能够在快节奏的创业般的环境中埋头苦干之人。公开交流、享有自主之权、创新、团队合作和客户成功是团队的根基。
>
> 工作职责：
>
> - 将客户问题描述转换为具体的需求；
> - 建议、设计、开发并实现新的特性和增强功能；
> - 生成高质量的需求文档、功能文档和设计文档；
> - 与组织中的开发人员和开发经理达成良好的合作。
>
> 职位要求：
>
> - 擅长 C 语言编程；
> - 注重结果；
> - 既能融入团队，也能独立工作；
> - 具有出色的写作能力和口头表达能力。

我们从最明显的地方开始说起：这则招聘公告中，大部分内容都是没有意义的，比如“注重结果”“既能融入团队，也能独立工作”和“具有出色的写作能力和口头表达能力”。现在问你自己一个问题——会不会有人看到了其中的一句话，然后心里想着：“注重结果？算了，我猜是不能申请了。”这些都是空洞的句子，没有什么作用，我敢打赌你在通读第一遍的时候会完全跳过。这样的职位公告也无法让你的公司从竞争者中脱颖而出，一家公司的“软件工程师”的职位公告和其他公司的并无二样。不妨想想，这些公司花

了数百万美元去做市场推广和品牌宣传，为的是让自己从其他竞争者中脱颖而出，但他们的招聘公告却非常一致。最后，最大的问题还是大部分招聘公告都是自私的。他们关注于公司想要什么，全是需求和职责的长长列表，却几乎忘光了应聘者的需求。特别是这则招聘公告并没有为潜在应聘者回答最为重要的问题：我究竟一开始为什么要申请这份工作？

如果你打算使用求职榜，就要做一份出色的招聘公告，需要把它当作一则广告去考虑。你不仅要解释这份工作是什么，更重要的是解释为什么应聘者要申请这份工作。你要避免标准的人事经理的那一套言辞，避免无聊地列出各种需求，要引导公司内部的“广告狂人”做出有创意的招聘公告来。另外，你还要把你们的工作广告放到程序员经常去的地方，比如 Reddit、Hacker News 和 Stack Overflow。最好的做法是，反过来使用求职榜：不寄希望于求职者找到你的招聘公告，而是花时间找到求职者并直接和他们取得联系。被动的求职者不会花时间去寻找工作，但他们对于找上门的新机会是持开放态度的。

11.3.4　专职招聘人员

在创业的早期，应聘者主要是通过推荐得到的。创始人和早期员工应该亲自去接触各自的人脉，吸引他们过来面试。然而这是很费时间的事，每个人都需要亲自参与可能意味着此时是不需要专职的招聘人员的。

随着公司规模的扩大，招聘过程也会越来越花时间。如果你已经邀请过所有认识的人，剩下的就只能去网上寻找。随着公司需要填补的空缺职位增多，你也必须花更多的时间去筛选应聘者，和他们联系，一个个去了解，安排面试日程，提出录用，核实他们的个人资料，这一切都要通过申请者跟踪系统进行组织。这时候就需要有一个全职的招聘人员，让技术人员解放出来，只关注于面试的过程（将在下一节讨论）。

一般来说，尽管公司内部配备招聘人员的成本更高，但却是更好的做法。他们会让你的创业公司能够持续生存和呼吸，所以他们能够成为公司更好的鼓吹者。此外，你也可以和他们紧密配合，定义出职位的精确需求或者你所寻找的应聘者的类型。例如在 LinkedIn，我会定期和招聘人员坐在一起，向他们讲述我们所使用的技术、我们所寻找的人才类型（现有的员工通常是不错的例子）、公司文化以及过去为了留住应聘者而采用的一些有效策略。我还会和他们讨论如何浏览简历、LinkedIn 个人资料、GitHub 个人资料以及 Stack Overflow 个人资料。不要跳过这些步骤，因为做 HR 的人和搞技术的人在阅读简历上是有很大差异的（Steve Hanov 在他的博客上专门举了很好的例子）。

11.3.5　过早优化

你可能听说过一个关于招聘的经典笑话。

> 老板把今天收到的简历的一半拿出来，扔到垃圾桶里，说道：“我不会招聘运气不好的人。”

这是一则有趣的轶事，但大家实际上不会这样做，对吗？好吧，大家只是不会故意这么做。我下面将列出几个招聘中的“行业标准”，它们和扔掉一半简历没有太大的不同，都是就是一种过早优化，出于错误的原因而排除了一些应聘者。

1. 拼写错误

如果你招聘的是文字编辑，因为出现了拼写或语法错误就把简历丢一边还可以理解。如果你招聘的是程序员，这是一种不好的直观推断。这并不是说程序员不应该了解语法或者拼写（他们应该了解），而是因为细小的拼写错误是随机发生的。每个人都会犯拼写错误，不管你多努力，你都无法在自己的文稿中把它们找出来（问问我的编辑就知道了）。

2. 糟糕的定位

你在博客、推文、求职页面以及工作广告的宣传标题中展示公司的方式，决定了你会吸引什么样的应聘者。如果你想在各种类型的应聘者中招聘到想要的，最快的方法就是把自己宣扬为一家针对程序哥[3]、忍者[4]和Unix大胡子[5]的公司，并会把在公司的所有时间花在对代码的迷恋[6]和咕噜咕噜大口喝着啤酒这两件事情上[7]。

> 我做的另一件事情就是开始用朴实无华的英语去宣传。我们尝试过放上吸引忍者、摇滚明星等人的广告，但是我发现这样的广告就等同于宣扬喝着干马丁尼酒的白人的那套文化——不喝干马丁尼酒的白人就不能从事这份工作。那么刊登广告寻求忍者和由于你不喜欢不幸运的人而扔掉一半的简历就没什么不同了。无论哪种方式，最终得到的都是更少的简历。
>
> ——Reginald Braithwaite，GitHub 开发人员

3. 大学学历

许多公司会把那些没有大学学位的人的简历扔掉；有些需要计算机科学的学位，有些则要求特定的GPA；有些只会看挑出来的几所大学而忽略其他的大学。虽然这听起来像是一种有效的主观式推断，但它仍然是一种过早优化。例如，Google 通过对他们面试和招聘的数以千计的人员数据进行研究后发现，把 GPA 和考试分数作为招聘或者预测工作表现的依据是毫无价值的——不过刚从大学毕业的学生除外，GPA 和考试分数和这些人有些许相关性。顶尖大学的计算机科学学位也许可以帮你写出更好的代码，但要对此进行检验只需检验他们的编程能力即可，而不是看有没有学位。

一些最有名的程序员和创业公司创始人，比如马克·扎克伯格、斯蒂夫·沃兹尼亚克、保罗·艾伦、比尔·盖茨、拉里·埃里森和迈克尔·戴尔，从来没有获得过大学学位。但这也并不意味着大学学位是没有价值的。不过，仅仅因为他们没有列出“合适”的大学学位，就把人家的简历丢掉，并没有比随机丢掉简历好多少。

4. 如何避免过早优化

我在采访 Gayle McDowell 的时候，发现了一种不会过早排除出色应聘者的最佳技巧。

> 人们犯的一个错误就是会在简历上寻找特征列表，找到缺少的一项——也许只是没有列出参与的外部项目，就会认为“这个人肯定对软件不是非常有热情”，

注 3：原文为 brogrammer，该词由 bro（兄弟）和 programmer（程序员）两个词演化而来，意指一些喜欢社交、擅于玩乐的程序员，与过去书呆子式的程序员形成鲜明对比。——译者注

注 4：指一些默默无闻、毫不声张的编程高手。——译者注

注 5：指 Unix 高手。——译者注

注 6：实际的招聘广告可能是：“想要快活快活并沉迷在代码之中吗？某某公司正在招聘。”

注 7：黑客马拉松的实际广告是：“还想喝啤酒？我们亲切的（女性）员工会在活动中为你备好。”

> 然后把这个人删掉。问题是这个人也许参与了许多外部项目，只是不知道要把它列到简历上。我之所以有这样的观点，是因为我和许多技术人员共事过，意识到他们在写简历这方面有多么糟糕。
>
> 所以我的建议是要反过来采用**寻找闪光点**的方法。这些闪光点可能是从好学校获得的出色的 GPA，可能是工作之余做过的很酷的项目，可能只是工作之余做过的很多小项目，也可能是他们在现在的雇主那里获得的奖励——我们应该只去看这些闪光点。
>
> ——Gayle Laakmann Mcdowell，Careercup 创始人、CEO

我们要寻找这样的闪光点。如果找到了，通常就足以开始面试了。

11.4 面试

> 我们在多年前做过一项研究，想确定 Google 的人是否都特别擅长招聘。我们研究了 1 万次面试以及进行面试的所有人，还有他们对应聘者的评分，以及这些人最终在工作中的表现，后来发现其实一点关系都没有。这完全就是一件随机的事情。
>
> ——Laszlo Bock，Google 人事高级副总裁

面试并不是完美的。面试官通常只有一个小时的时间去决定他们是否要在接下来的几年和这个人一起工作。在这样短暂和人为的过程中，几乎不可能获得足够的信号。理想的面试也许根本就不是面试：你只需要聘请这个人工作几个星期，看看他表现如何再做决定即可。但这通常是不切实际的，所以面试过程就是一种妥协，最多也只能被看作是一个压缩和失真的过程。

面试官甚至更不完美。一个著名的例子来自于管弦乐团的试奏面试。在 19 世纪 70 年代，只有不到 5% 的乐团成员是女性，人们一般认为女人的表现不如男人。在 19 世纪 80 年代早期，管弦乐团开始采用拉帘试奏，演奏者在幕布后面表演，评委只能听到音乐而无法看到是谁在演奏。因此，到了 90 年代末期，女性成员的数量增长到了 35%，上涨了 7 倍。另一个例子来自 2010 年对假释法官的研究。研究表明，犯人被同意假释的概率明显受到他们的假释聆讯发生在一天中的什么时候的影响：早晨大概是 65%，在正午之前会渐渐下降到接近 0%，午饭之后会跳回到 65%。可以说在假释决定中，法官的胃比呈现的任何证据都要重要得多。

从中我们可以引申出两个含义。第一，也许要避免在午饭之前面试应聘者；第二，更为重要的是，我们必须认识到面试过程有时会导致不正确的决定。我们偶尔会拒绝一个出色的应聘者或者接受一个糟糕的应聘者。一些好的公司已经认识到面试的缺点而慎之又慎——如果我们招聘到错误的人，还要花时间去培养他们，弄清楚他们为什么不能努力把工作做好，出了问题还要帮他们救场，把他们列入绩效考核计划中，最终将他们扫地出门。经过这一切之后，我们又得花更多的时间去招聘他们的代替者。解决因招聘到糟糕的应聘者而带来的问题，比偶尔错失一个出色的应聘者要痛苦得多。所以，除非你绝对有把握，否则在招聘的时候请说“不”。

记着上面我说的，再来看看面试的基本构成。

11.4.1 面试过程

1. 第一步：联系

在本章之前的内容中，我讨论了如何找到出色的候选人，而下一步就是和他们取得联系。我们可以通过 email、LinkedIn 消息甚至发送 Twitter 等方式，亲自和他们联系。当然很多都不会有回应，而有回应的人中大部分也都不感兴趣，只有少数人仍有希望。我们的目标就是让他们对在你的公司工作感兴趣，然后安排他们进行电话面试。

2. 第二步：电话面试

我不喜欢讲电话，但是电话面试又是一种必要之恶。现场面试成本高昂，在金钱（飞机、酒店、汽车、饮食）和时间（每次现场面试，你的公司至少有一至两个人无法工作）上都是如此。因此，电话面试的目标就是把可能会通过现场面试的候选人给过滤出来，这意味着电话面试在问题和难度上应该和现场面试几乎是一样的，但是应该更简短一些。

在大多数公司，两到三次电话面试是标准的做法。第一通电话通常是由招聘人员或者招聘经理描述公司的情况、职位角色，大概了解应聘者想要的是什么。假设应聘者的关注点和公司的关注点是一致的，我们就可以安排第二次和第三次电话，这就是正式的技术面试，可以由创始人、招聘经理或同事进行。为了能够在电话面试中用上代码，我们可以使用一些在线协同编辑器，比如 Google Docs、CollabEdit 或者 Stypi。

3. 第三步：现场面试

如果应聘者在电话面试中表现出色，下一步就是让他们来到现场。他们应该与公司的 4~8 个人会面，这样通常可以降低偏见的影响，但又不会让应聘者负担过重并使面试时间过长。每一名面试官都可以在以下方面各有侧重：沟通技巧、文化契合、编码、系统设计和亲切感。

在创业初期，创始人应该对每一名应聘者进行面试，因为早期员工对公司的成功与否有着巨大的影响。此后，随着公司规模的扩大，面试官应该主要从应聘者面试的团队中选择。

每一名面试官从面试出来时要么得说“可以”，要么得说“不可以”，不允许说“也许吧”。只有一个“可以”而没有其他团队成员的同意，照样不行。只有获得所有面试官的“可以”的应聘者才会被录用。录用一个有问题的应聘者比错过一个出色的应聘者更糟糕，所以即便只有一个“不可以”通常也足以拒绝一个人。唯一的例外是如果有人过去曾经和那位应聘者亲密共事过，并强烈为他打包票。现实的工作经历当然比在人为的面试中得到的一个“不可以”更有说服力。

11.4.2 面试问题

面试更多的是艺术而不是科学，但是在电话面试和现场面试中，可以遵循一些基本的原则，让面试官和应聘者的时间得到最好的利用。好的面试可以让应聘者有机会去教、去学、去展示他们的技术能力。

1. 让应聘者教你一些东西

不要把面试等同于质问。许多面试官会直接跳入到技术问题中，直接用智力题和编程难题去狂轰滥炸，甚至连招呼也不打。你在那儿不是要拷问应聘者知道多少，也不是要让应聘者承

认自己是个糟糕的码农。他们不是你的敌人，面试本身就是有压力的，所以请善待他们。

我们要把面试当作两个人之间有礼貌的交谈，介绍你自己，简短地谈谈你是做什么的以及想要填补的职位类型。询问应聘者他们是做什么的，是否需要水或者休息一下。之后，和任何愉快的对话一样，把话题转移到应聘者身上，尝试去了解他们。让他们告诉你一些与自己有关的事情，谈谈他们过去从事过的项目，描述他们希望在以后能从事的项目，了解他们为什么对你的公司感兴趣。

大多数人喜欢讨论自己，所以这也是让应聘者轻松进入面试的好方法，更是了解他们的经历、激情以及目标的好方法。面试官（是招聘经理则更好）至少应该把大部分时间花在讨论这些话题上，在结束面试时，应该能很好地感觉应聘者是否适合该职位。

面试也能够很好地了解应聘者的知识深度、对之前工作的热情以及沟通能力。深入下去，多问问题。是不是弄清楚了他们过去完成的工作？他们所做的东西、所用的技术、现实的方式是否已经清楚了？理想的情况下，你应该让应聘者教你一些新的东西。这就是 Sergey Brin 在 Google 面试中采用的策略。

> “我打算给你五分钟，”他（Sergey）告诉我，“当我回来时，我想让你向我解释一些我还不曾了解的复杂的东西。”然后他就离开房间朝着点心区走去。我看着 Cindy，她告诉我说：“他对所有东西都很好奇。你可以谈谈爱好，谈一些技术的东西，什么都可以。只要确保那是你真正深入理解的就行。”
>
> ——Douglas Edwards，Google

你甚至可以让这一过程变得更加真实，让应聘者做一个汇报展示。你还可以让他们教整个团队，而不是只教面试官一个人。

> 我们所从事的业务更多是社会化的，而不是技术上的，它需要更多地依赖员工之间的沟通，而不是他们和机器的沟通能力。所以招聘的过程至少需要关注一些社会化和人际沟通的品质。我们发现最好的方法其实是让应聘者去试一试。
>
> 这一想法再简单不过，就是让一个应聘者准备 10 或 15 分钟的个人展示，介绍过去工作中让你们团队感兴趣的一些东西。可以是新的技术、是第一次体验这种技术的经历、是一门历经艰难学到的管理课程，或者是某个特别有趣的项目。应聘者选择了主题，可能要经过你的认可。日期确定之后，你可以聚集起一小群听众，由那些即将成为新员工同事的人组成。
>
> 当然，应聘者会紧张，甚至可能不情愿经历这样一个过程。你必须解释清楚所有应聘者都会对这样的试讲感到紧张，给出你的理由，只要坚持一点：为了了解每个应聘者的沟通技能，让未来的同事参与到招聘过程中。
>
> ——Tom Demarco、Timothy Lister，《人件》

2. 让应聘者有机会去学东西

虽然你是在面试应聘者，但他们其实也是在面试你。他们也希望了解公司是干什么的，他们会扮演什么样的角色，你们使用什么技术，他们将和谁一起工作。每一位面试官都应该为应聘者留出一些提问的时间，让他们可以了解你的公司。另外，至少有一位面试官，最好是招聘经理，应该带应聘者参观一下办公室，并大概介绍一下你们在公司的工

作。让他们了解你们使用的技术、你们做的产品、谁是你们的用户、公司的目标。无论是现场演示、代码详解、架构图、幻灯片和视频都是可以的，也可以谈谈你们的文化和使命（阅读第 9 章了解更多信息）。传递出你们的激情，看看是否能够感染他们。

> 我是做编译器的，我注意到我许多搞编译器的朋友都去了 Google 工作。我不知道为什么弄编译器的人要去这样傻傻的小搜索引擎公司，但我发了简历参加了面试之后，才感到这是我参加过最好的面试，感觉真是太棒了，就像跟一群疯狂且聪明的人在一起做技术即兴表演一样。这是我在参加资格考试之后经历过最激励人心的事情。我是一个古怪的人，会被资格考试弄得激动不已，但是这种精神上的挑战真的很有趣。
>
> Google 对面试的人超级体贴。他们从简历中看出我是做编译器的，所以安排在面试桌前面试我的六个人全都是做编译器的，全部都是。所以我立刻就和他们有了专业的沟通，我可以讨论我的研究，我们可以讨论共同了解的东西，气氛也得到一点缓解，等到进入某个问题时，我已经完全放松了。我们都是搞系统的人，想东西的方式大致是一样的。面试的体贴实际上就体现在此。
>
> ——Kevin Scott，LinkedIn 高级副总裁，Admob 副总裁，Google 主管

每一名面试官都是公司的大使，所以我们要选择聪明的人。在创业早期，我们没有足够的人可以挑选，所以每个人都必须参与所有的面试。但是随着公司的成长，招聘岗位可能要更加专业化。一些开发人员擅长进行技术面试并找出厉害的程序员；一些则是出色的“偷心者”，能够说服正在考虑多个职位的应聘者挑选你提供的职位；还有一些则更擅长通过博客、会议演讲和开源项目的品牌宣传方法，让应聘者申请你的职位。这一切和招聘同样重要，而且需要的是不同的技能，得花时间去培养。

如果我们不能专门为招聘留出时间并给从事招聘工作的员工以报酬，招聘就永远都是二等工作，员工永远不会花时间去写博客或者做开源项目，应聘者面对的也只是丝毫不能鼓舞人心的面试，这样得历尽艰辛才能打造出一支强大的团队。我们不应该把招聘当作是一项会剥夺“真正工作”时间、让人分散注意力的事，而是要把它列入工作日程中，和其他工作任务没什么两样。每一名开发人员都应该有具体的目标（例如挑选三位候选人或者每月写两篇博客），我们在进行项目计划的时候要为这些目标估算出时间并纳入员工考核。

为了避免招聘任务对工作造成过大的影响，我们不妨让每一名开发人员一次只关注一点。例如可以设立技术面试专家团队，让他们每个星期奉献五个小时进行编程面试；设立说服专家团队，每星期投入五个小时说服候选人加入到公司中来；而品牌宣传专家团队则每个星期要投入五个小时去写博客、维护开源项目和做演讲。我们可以定期让这些招聘团队的工程师进行轮转，帮助他们培养新的技能（阅读第 12 章，更详细地了解如何投入时间学习）。

3. 让应聘者展示他们的技术能力

技术面试的行业标准就是让应聘者解决计算机科学基本的数据结构和算法问题，预先想出整个解决方案（而不是迭代式的），并把思考的过程说出来，然后再写到白板上，没有任何语法高亮、没有自动完成、没有 Google、没有 Stack Overflow、没有编译器、没有开源库、没有文档、没有测试、没有简单的重构方法。过早优化是所有罪恶的根源，而不只是在面试中。我们要求面试者去优化个别该死的代码路径，没有任何分析工具，而且

通常都是以代码的清晰和简单为代价的。如果有人在现实中像这样去写代码，你可能已经把他炒掉了，所以为什么要用这样的方式去面试？

智力题就更糟糕了。咨询公司和一些软件公司，最著名的是 Google，已经使用了好多年，但是没有证据表明，这些题和实际的工作表现有任何相关性。

> 站在招聘方的角度，我们发现智力题完全就是浪费时间。一架飞机可以装入多少高尔夫球？曼哈顿有多少煤气站？完全就是浪费时间。这些题无法预测什么，他们主要就是为了让面试官看起来聪明而已。
>
> ——Laszlo Bock，Google 人事高级副总裁

有一些更高效的方式可以了解应聘者的技术能力，并能让应聘者和面试官有更好的体验。下面是一些现实中的例子。

带上自己的笔记本电脑（bring your own laptop，BYOL）

可以让应聘者带上自己的笔记本电脑，配置好他们喜欢的编程环境。另一种替代的方法是问他们用什么 OS 和编程工具，提前为他们准备好笔记本。如果你想了解他们实际的编码水平如何，可以让他们使用 Google、Stack Overflow 和其他一些他们在日常编程中经常使用的技术。

> 在 Coursera，我们会进行现场的编码操练：你把自己的电脑带来并用它解决一个问题——我想这样比在白板上编程好很多。我们会有分级规则，精确记录他们的表现，所以我们知道哪些应聘者是不同寻常的。
>
> ——Nick Dellamaggiore，LinkedIn、Coursera 软件工程师

带回家去挑战

给应聘者提供一些问题，让他们回到自己舒适的家中去解决问题。当他们到现场面试的时候，和他们一起对代码进行评审。带回家去挑战不应该成为面试的必备环节，也不是所有应聘者都有时间这样做，它不应该是你进行的唯一的技术面试，因为一些应聘者会作弊，找他们的朋友来帮忙。但是对于那些面试时会紧张的应聘者来说，这也是个很好的选择，因为这样有机会让他们在一种更加自然的编程环境中展示自己的技能。

> 我们在 Pinterest 尝试过在面试之前先举办一场编程挑战赛，主要是在早期没有足够的技术人员去开展面试的时候。我们让他们去参加编程挑战，既可以很好地检验他们对公司是否有兴趣——如果他们不乐意花三个小时去进行编程挑战，他们可能对此并不感兴趣，对我们的团队来说也是了解他们技术能力的一种高效方式——可以在方便的时候异步评审代码。如果代码看起来很糟糕，我们可以马上发送一封拒绝邮件，不会在这名应聘者身上花更多时间。
>
> —— Tracy Chou，Quora 和 Pinterest 软件工程师

现实问题

我们可以不在面试中用一些设计好的计算机科学的基本问题，而是让应聘者解决一些你公司实际在做的东西。这样会让你更好地知道应聘者在现实中会有怎样的表现，也让应聘者更好地知道你们在公司中做的东西。

> 在 Jawbone，数据副总裁 Monica Rogati 会交给应聘者一份数据资料，给他们三个小时的时间浏览，然后让他们说出他们的发现。“这种方法能测试我希望在数据科学家身上看到的四种关键品质：技术水平、数据创造力、沟通技能以及是否是结果驱动型的人”，她说。同时这也能让应聘者一睹公司的工作日常。
>
> ——Samantha Cole，Fast Company

> 如果我正在面试你，差不多就会这样说：“这是我们这个星期一直在处理的问题，接下来我会把它交给你，我真的还不知道要怎么解决，你看看你会怎么做？”然后就站到一边。如果是进行技术招聘，我们会找一个以后将和你在一个团队的人，一起在 GitHub 上结对编程。
>
> ——Zach Holman，GitHub 软件工程师

结对编程

结对编程的方法就是假装你已经录用了应聘者，现在是和他们工作的第一个星期，你正在帮助他们提高能力。和他们一起坐在电脑前，对一个真正的项目进行结对编程，权衡一下谁负责编写代码，谁负责操纵、观察。你能够了解他们可以多快地学习，如果你录用了他们，他们能够写出什么样的代码。应聘者则能够近距离了解产品、所涉及的技术，以及工作的日子通常是怎么样的。

> 我招聘了一个年轻人，他早上就过来面试，那时公司只有我一个人，于是我就说：“你好，很高兴见面，请坐，我们今天来结对编程吧。”在他那天离开之前，我告诉他被录用了。这是我做过的最好的招聘，我想这是一个很好的过程。
>
> ——Dean Thompson，NOWAIT 公司 CTO

让应聘者上一天班

你可以把结对编程的概念进一步引申，让应聘者正常地上一天班来代替面试。给应聘者一台电脑和一位导师，让他们检查代码，让代码跑起来，做些修改，和团队一起，参加会议。虽然需要做一些后勤保障工作，但对你来说也许是了解应聘者在工作中会有怎样表现的最好方法；对应聘者来说，这也是了解在你公司工作感觉如何的最好方法。

> 我们通常会让这些人进来，有时候要待几天，和他们一起工作。实际上，我们是要让他们坐下来去真正处理几个问题、参加会议和头脑风暴会，这样可以了解他们能够在多大程度上参与进来。第一天有时可能很吓人，但是几天之后，他们就开始放松了，你可以看到他们是什么样的人，他们是否参与进去了，他们是否充满热情。
>
> 我其实不相信测验，但我相信和应聘者一起编程。我不使用修饰过的编码练习——通常都太虚假了。我们使用真正的代码，通常来自项目的核心代码库。
>
> ——Jonas Bonér，Triental Ab 和 Typesafe 联合创始人

11.5 录用

如果你找到了一位所有面试官都说“可以”的应聘者，你就要尽快行动了。第一步就是与证明人核实。应聘者会提供他们自己的证明人，你应该使用这些关系，但也别畏惧去找找自己的人脉，看看有没有谁以前和这位应聘者一起工作过（阅读 10.1.1 节了解更多信息）。你的主要目标是验证应聘者的简历，但更为重要的是，了解和他们一起工作是什么感觉。假设你考虑提供一个工作机会给 Anna，下面是可以问 Anna 的证明人的一些问题。

- 你怎么认识 Anna 的？你们在一起工作过吗？有多长时间了？
- 告诉我 Anna 的职业。她最出色的成就是什么？
- Anna 最大的优势是什么？她正致力于改进的是什么？
- Anna 在这份工作相关的技能方面有些什么样的经验？
- 和 Anna 一起工作感觉如何？你想再次和她一起工作吗？
- 为什么 Anna 要离开她当前的工作？她希望得到什么？
- Anna 是和你共事过的最出色的 1% 吗？最出色的 10% 呢？

如果证明人的核实通过了，就要尽可能快地给应聘者提供录用。不要超过一两天，才能让应聘者知道你对他们是感兴趣的，那时他们也还清晰记得你的公司。如果等待的时间太长，他们可能就会答应其他地方的录用机会，因为几乎所有出色的应聘者都会同时面试多家公司。记住，速度制胜。

> 所以，我们的创始人之一 Chris Wanstrath 发出了一条推文问道：“有人了解 Ruby 吗？”我的回答是：“是的，我了解 Ruby！”后来 Chris 问道：“有人了解 Java 吗？”我又回答：“我在学校用过 Java，我稍微了解点 Java。”Chris 紧接着说：“把你的简历 email 给我。”我把自己的简历发了过去，不久之后，他问：“你想喝点咖啡吗？”我回答：“当然。”
>
> 大概在会面的 10 分钟之前，他们又问：“要不还是去酒吧喝点东西算了？”我回答：“OK，酒吧面试，可以啊。”
>
> 我们谈了大概半个小时，然后我去了趟洗手间。当我回来的时候，他们对我说：“好了，你被录用了，5 分钟内告诉我‘可以’还是‘不可以’。”我想他们是在开玩笑吧，但我还是说：“可以，没问题。”过程就是这样了。
>
> ——Zach Holman，GitHub 软件工程师

你要复制的也许是 GitHub 的精神，而不是其做法。关键点不在于要在酒吧中面试——Zach 的故事只是一个例外，这并不是 GitHub 正常招聘的方式，但是却找到了一种可以让应聘者对面试和公司感到兴奋的做法，并在这种兴奋消退之前给他们录用的机会。你应该亲自提供这种录用机会或者通过电话进行，因为这样才有机会祝贺应聘者，表明让他们加入团队的热情，再次让他们知道自己对公司使命是有所影响的。此后，紧接着要做的就是通过 email 发送书面的录用通知。

11.5.1　应该提供什么

你如何才能提供给应聘者无法拒绝的录用机会呢？一个词：倾听。

> 我和别人第一次交谈时会问：“你在寻求什么？你想要什么？”这不是我在推销 Tindie，我只是在倾听他们。在交谈中谈得越多的人一般都越觉得这是一次更好的谈话，所以我会倾听——我是真正地倾听。在尝试销售之前，我会问“你想要什么”，因为如果你先去销售，然后再问，人们通常只会说一些你想听到的东西。
>
> 我发现有些人并不理解在一家处于早期阶段的公司工作是什么样的。他们所设想的也许是 Google 或者 Facebook，此时这种感受就很明显了：请勿见怪，那并不是我们。我会描述 Tindie 是什么样的公司，在会谈结束的时候，我会说：“在这次会谈之后，你应该花些时间想想是否愿意进行面试。”
>
> 这不是最灵活的方法，但对我们一直都是非常有用的。当我们到了要提供录用的时候，还从来没有谁不接受。我会真正努力了解他们需要什么，他们在寻找什么。
>
> 我总是会问他们，如果有三个工作机会摆在你面前，你如何决定要选择哪一个。答案通常非常能启发人。我听过最多的答案就是“我不知道”，所以我会继续引导他们，大多数时候都能从回答中洞察到什么对他们才是重要的，他们的价值观是什么，如果在 Tindie 工作，Tindie 对于他们所要的可以提供什么（不仅在薪水方面，还有经验、与业务相关的东西，等等）。
>
> 我面试过的一个人说道：“好吧，我必须和妻子谈谈，因为‘我在哪里工作’是一个家庭决定。”我就说：“你想让我和你妻子谈谈吗？”他说：“是的！”接着我就和他妻子谈了话，不是我在面试她。但也知道了他有两个孩子——他有家庭，他不是唯一做决定的人，还有其他的因素在起作用。我想这也是后来他会接受这份工作的原因。
>
> ——Julia Grace，Weddinglovely 联合创始人、Tindie CTO

许多公司把薪水放在第一位，但这只是尝试说服应聘者加入公司时可以用来讨价还价的一项。好的工作招聘应该说清楚机会、薪酬和福利。

1. 机会

机会是一个工作机会中最重要的部分。我们提供的职位应该是一个参与更大的使命的机会，是和了不起的人一起工作的机会，也是对事业有所发展、有更大的影响力、从头开始做一些东西的机会（阅读第 9 章，了解有关使命及文化的内容）。我们应该在招聘过程的每一个环节不断提醒应聘者这些东西：在职位描述中、在最初的 email 中、在电话面试期间、在现场面试期间。我们也应该让录用信涵盖这些内容，不要写一份无聊的法律文档，而是写一封个性化的、鼓舞人心的信。例如，Apple 的录用信，和大多数 Apple 的产品一样，都是放在一个设计精美的白色包装中，有着优雅的文字排版，写着以下内容。

> 有一种工作只是工作，有一种工作却是你终生的追求。
>
> 这种工作草木皆情，由你全情打造。这种工作会让你从来不妥协。这种工作也会让你甘愿牺牲周末。你可以在 Apple 找到这样的工作。在这里的人们不会闲

庭信步，他们到这里击水三千。

他们希望自己的工作能有一些不同的东西。

一些重要的东西，一些不可能在其他地方得到的东西。

欢迎来到 Apple。

——Paul Horowitz，OS X Daily

老实说，你更愿意看到这样的录用信，还是充斥着给所有员工看的一大堆法律术语以及在什么情况下会被解雇的信？

2. 薪酬

我们应当提供一个薪酬方案，把会谈中商定的薪酬写下来。为此，这个薪酬方案必须是公平且透明的。公平意味着这份薪酬足以让应聘者过上舒适的生活，在市场上也是有竞争力的。公平也意味着这个职位和公司的其他员工相比也是有竞争力的。这也是透明的缘由——透明产生信任，如果每一名员工都知道你是如何确定薪酬的，他们就不用担心旁边做同一份工作的同事收入可能是他们的两倍。

实现公平和透明最简单的方法就是使用公式去计算薪酬。你可以在给应聘者看具体的薪酬数字之前说说这个公式，这样他们就能够准确地知道你是怎么算出这个数字的，也知道这个公式会用在公司的每个人身上。虽然无法保证应聘者会对这个数字满意，但至少他们会感到是被公平对待的。你的公司必须根据你们的价值和从事的商业类型找出自己的计算公式。举个例子，计算薪水最简单的公式就是一张根据级别提前定义好的薪水表格，如表 11-1 所示。

表11-1：根据级别提前定义的工资表

级别	工资
初级	10 万美元
高级	15 万美元
主管级	20 万美元

你可以使用在线薪水计算器计算出当地每一个级别有竞争力的数字。更复杂的做法是把其他一些因素也纳入考虑，比如职位类型、地点和经验。下面是一家叫 Buffer 的创业公司使用的公式：

```
工资 = 职位类型 × 级别 × 经验 + 地点（+ 1万美元，如果选择薪水的话）
```

Buffer 对于不同的职位类型有着不同的基本工资（例如工程师是 6 万美元，执行官是 7.5 万美元），不同的级别有不同的系数（例如副总裁是 1.1 倍，CEO 是 1.2 倍），不同的经验也有不同的系数（例如初级是 1 倍，高级是 1.2 倍），还有生活成本的调整（例如奥斯丁是 1.2 万美元，旧金山是 2.4 万美元），还可以选择多 1 万美元的薪水还是多 30% 的股份。所以，如果你用这条公式给一个居住在旧金山，选择要更多薪水而不是股权的初级工程师提供录用机会，算出来的数字就是：

```
工资 = 6万美元 × 1.0 × 1.0 + 2.4万美元 + 1万美元 = 9.4万美元
```

股权的计算公式都建立在你何时加入公司（阅读 10.3.2 节，了解股权的入门知识）。后来

的员工承担更少的风险，所以他们得到的股权也更少。最简单的方法就是使用预先定义好的数值表格。读者可以查阅 http://www.hello-startup.net/resources/equity，表 11-1 列出了给予创业公司员工股权的通常数量，这些数量是根据员工的职位、级别、员工数量给出的。另外，也可以使用公式去计算。

有一条著名的公式叫股权等式。它的思路是如果你认为某个人（员工或投资者）可以提高公司的平均输出，你愿意将公司的 $n\%$ 给他，使得你剩余的部分（$100 - n\%$）比以前的 100% 更有价值。这个思路可以用下面的等式来表示：

$$n = \frac{i-1}{i}$$

其中 i 是你招聘了这个新人之后公司的平均输出。例如，如果你招聘了一个新程序员，你认为他可以将公司的价值提升 20%，那么 $i = 1.2$，$n = (1.2 - 1)/1.2 = 0.167$。也就是说，你最多可以给这名程序员公司的 16.7% 而仍然不赔不赚。当然，你还要付给这名程序员薪水和福利。如果这些加在一起假设是 10 万美元，而公司价值 100 万美元，那么 10 万美元就是 10%，你能给的就是 16.7% − 10% = 6.7%。考虑应聘者可能会谈判而且你不仅要不赔不赚，还要有收益，你也许要修改下这个数字，最终在实际提供时差不多就是 3%。

按照经验，给员工的股权应该比给投资者的股权更大方一些。投资者通常只会在一开始的时候给公司增加价值，有时候还会因为过早退出而给公司的长期运营带来伤害，而员工通常会持续为公司提供越来越多的价值。为员工提供更多股权符合你的利益，因为员工为公司创造的价值越多，他们为自己创造的价值也就越多。

此外，在招聘方面，大多数创业公司都无法在薪水方面和大公司竞争，所以你只能提供更多的股权来弥补。你可以把它作为一种投资的机会提出来。例如，如果应聘者因为加入你的创业公司而没有加入大型、成熟的公司而导致每年在薪水上损失 1 万美元，那么在 4 年之后，他们就投入了 4 万美元（因为上税的关系，实际会少于这个数）去换取（假设）3% 的回报（加上许多无形的东西，比如创业经历、学习机会和新的关系）。如果公司最终价值 1000 万美元，那么 4 万美元的投资大概价值 30 万美元（这里做了很多简化，阅读 10.3.2 节了解更多信息）。当然，大部分创业都是失败的，这点对应聘者没什么好隐瞒的，要告诉他们这是一种高风险的投资。在创业公司，特别是在早期阶段，你要招聘的正是那些能够坦然接受风险的人。

3. 福利

在福利方面要有创造性，你有时可以提供相对便宜但对员工极有意义的东西。例如，提供额外的一周假期只会花掉公司大约 1/50 的应聘者薪酬，差不多相当于提高 2% 的奖金。但这种能够和心爱的人在一起更多时间的福利，对应聘者来说却远远胜过那 2% 的奖金。了解这种需求的唯一方式就是倾听。

下面列出了一些可以提供的福利的思路：

- 保险（健康、牙齿、视力和生命）；
- 假期（度假、节假日、病假）；
- 食物（免费早餐、午餐、晚餐、点心和饮料）；
- 薪酬（报销安家费，401K 缴存比例，奖金）；
- 工作现场福利（日托、干洗、按摩、汽车修理）；

- 健康（报销健身会员费用、运动课程、运动队）；
- 日程（支持灵活的工作时间和在家办公）；
- 活动（提供远足团队、阅读小组和志愿者小组的资金）；
- 学习（提供图书、课程、举办演讲及参加会议的资金）；
- 通勤（列车月票、地铁月票、停车月票、班车）；
- 自主权（黑客马拉松、20% 时间、孵化器）；
- 硬件（强大的笔记本电脑和台式机、大屏幕、平板电脑）；
- 工作环境（私人办公室、舒服的椅子、可带宠物）。

11.5.2 跟进和谈判

大多数应聘者都需要一些时间去做决定，但是不要让这个过程拖得太长。时间是交易的杀手，所以要安排定期跟进，比如让招聘经理打电话给应聘者，了解他们最近怎么样，问问他们是否有任何问题。让应聘者和员工、CEO、CTO 或者任何可以帮助他们做决定的人共进午餐。或者，给应聘者送上一份关怀。当我在考虑 TripAdvisor 的职位时，他们给我发了一件抓绒衫和一张漂亮的卡片。这可能值不了多少钱，但是却让我对这家公司的感觉大为不同，到现在我依然记得这件事。

如果应聘者对职位有兴趣，他们也许会对你提供的条件进行谈判。不要觉得被冒犯了：这是面试过程正常的一部分。你应该提前知道你们的限度：在薪水上可以做多大的让步、可以多提供多少期权、可以给出什么样的福利去达成交易。如果你们使用公式计算薪水和股权，就提前告诉应聘者“为了保持透明和公平，你们对公司所有人都使用相同的公式，这些数字是不可谈判的”（然而你仍然可以对福利进行灵活的处理）。你也许会失去一些应聘者，他们会去其他开价比你高的公司；但你也会因为营造更加公平和透明的职场而收获更多，特别是对女性来说，她们不可能像男性那样为一次录用机会过多谈判。

11.6 小结

2014 年 11 月，Apple 成为历史上第一家估值超过 7000 亿美元的公司。Apple 因其优雅的设计、精湛的市场推广、前沿的硬件、集成的软件、高效的供应链和对保密的痴迷而极富声誉。但是史蒂夫 • 乔布斯——Apple 已故的联合创始人认为这家公司最重要的任务和最伟大的才能是什么呢？答案就是：招聘。

> 生命中大多数事情的关键点是，平均质量和最佳质量之间的动态范围，最大也就是二比一。举个例子，如果你在纽约，最好的出租车也许能比普通的出租车快 20%；在计算机领域，最好的 PC 也许比一般的 PC 好 30%。从量级上看并没有那么大的差异，你很少能找到二比一的差距，无论选什么做比较。
>
> 但在我关注的领域——最初就是硬件设计——我注意到普通人实现的东西和最出色的人实现的东西之间的动态范围是 50 : 1 或 100 : 1。还要考虑到，后者都是知道要追求精益求精的人。这就是我们的情况。你可以打造一支由追求 A+ 的队员组成的团队，一小队 A+ 的队员就可以远远超过由 B 和 C 的队员组成的庞大团队。这就是我们所要努力去实现的。
>
> ——史蒂夫 • 乔布斯

本章探讨了找出 A 级队员的一些方法，也谈到了要寻找谁（敏思笃行的合伙人、有无尽求知欲的通才、T 型专才），寻找什么（聪明并能把事情做好、良好的文化契合、出色的沟通技巧），但最重要的事情还是如何寻找他们。你可能在每一家公司都听到过这种陈词滥调："我们只招聘最好的 1% 的程序员。"停下来想一秒钟，这究竟意味着什么。每家公司都要招募顶尖的 1%，每一家都是这样。但是从数学上，你知道他们是不可能全都能实现的，无论他们面试的是多么出色的人。Joel Spolsky 举了一个很好的例子：想象在一种假设的情况下，你拿到一份招聘工作的 100 份申请。面试的过程很顺利，你可以从这 100 人中选出最佳的应聘人并拒绝掉剩余的人。这是不是就意味着你招聘到了顶尖的 1% ？其实并非如此。想想你拒绝的那 99 名程序员会做什么。

他们会寻找另一份工作。

下一次你或者其他人发布招聘公告的时候，可能收到的就是这 99 个被你拒绝的人的申请，外加一份新的申请，来自于这个世界上大约另外 1800 万程序员中的一员。即便你的面试过程依然出色，足以选出那第 100 位应聘者，但你实际招聘的仍不是顶尖的 1%，而是 1800 万程序员中随机的一名应聘者，大致也就是最好的那 99.99999% 的。

Spolsky 的见解有明显的夸张，对大多数公司来说可能真实情况远非如此。顶尖的 1% 的程序员很稀少，而申请工作的就更少了。所以如果你的招聘策略只是在线发布招聘公告然后等待申请，不管你的招聘过程做得多么好，几乎可以肯定，你无法找到现有的最出色的程序员。

最出色的程序员并不会上门来找你，你必须走出去找他们。你必须在你的人脉、开源产品和会议上到处撒网。想要让鱼儿上钩，你就必须通过博客、开源软件和演讲等方式建立自己的技术品牌，吸引出色的开发人员。你也必须让他们参加面试，提供和其他公司有所不同的工作条件，把他们钓上岸。这是你如何得到 A 级选手的方法，也是如何成为世界上最有价值的公司的方法。

> 我认为它（招聘）是最重要的任务。假设你自己在创业，现在需要一位合伙人。你要花很多时间去寻找合伙人，对吧？他就是你公司的 1/2。那为什么你就可以花更少的时间去寻找公司的 1/3、1/4 或 1/5 呢？当你创业时，前 10 个人将决定公司是否会成功。每个人都是公司的 1/10，所以为什么不花必要的时间去找到所有的 A 级选手呢？如果有 3 个人都只是一般般，那你要一家 3/10 的人都是一般般的公司干什么呢？与大公司相比，小公司对出色的人的依赖程度要大得多。
>
> ——史蒂夫·乔布斯

第 12 章

学习

> 在急剧变化的时代，唯有学习者才能继承未来。不再学习的人通常只能全副武装地生活在不复存在的世界里。
>
> ——Eric Hoffer，*Reflections on The Human Condition*

软件行业也许是有史以来变化最快的行业之一。自 2000 年以来，我们已经看到了几十种主要的新编程语言（例如 C#、D、F#、Scala、Go、Clojure、Groovy 和 Rust），旧的编程语言也出现了巨大的更新（例如 C++ 03、07、11 和 14，Python 2.0 和 3.0，Java 1.3、1.4、5.0、6.0、7.0 和 8.0），这些语言配套的数百种框架、库和工具也在发展着（例如 Ruby on Rails、.NET、Spring、IntelliJ IDE 和 Jenkins），出现了几十种新的数据库（例如 MongoDB、Couchbase、Riak、Redis、CouchDB、Cassandra 和 HBase），也出现了一些新的硬件平台（例如商用硬件、多核 CPU、智能手机、平板电脑、可穿戴设备和无人机），还出现了一些新的软件平台（例如 Windows XP、7、8、10，OS X 10.0 – 10.10，Firefox，Chrome，iOS 和 Android），以及一些敏捷方法论（例如 XP、Scrum、Lean、TDD、结对编程和持续集成），开源软件也出现了爆发（例如 Amazon EC2、Heroku、Rackspace 和 Microsoft Azure），等等。如果你在软件行业工作，每一年你的很大一部分知识都会被淘汰。

这就是为什么世界上最出色的软件开发者和软件公司都有一个共同点：都在孜孜不倦地学习。Erlang 编程语言之父 Joe Armstrong 说过，成为更出色的程序员的最佳方法就是“花 20% 的时间去学习——这是一种复利”。《精益创业》的作者 Eric Ries 把学习定义为“创业过程必不可少的单元”。在这本书的最后一章，我打算讨论每一名程序员和每一家创业公司如何才能安排出专门的学习时间，以及为什么要这么做。我会先介绍学习的原理，然后再转到三个最常见的学习技巧：研究、实现和分享。

12.1 学习的原理

在专业运动中，艰苦的锻炼和密集的训练课程是标准要求。同样，专业的音乐家、舞蹈

家和象棋选手每天也要花时间去磨炼他们的技艺。然而对大多数办公室工作来说，只要你从学校毕业进入新公司并通过了入职培训，专门用来学习的时间就结束了。换句话说，如果你在一般的办公室走一圈，看到的几乎每个人都是被困在瓶颈期——他们没有学习，也没有提升自己的技能。可以说除了其中的精英，每个人都是如此。

在《异类》一书中，作者 Malcom Gladwell 证明了在大多数学科中，精英型的成就者和其他人的区别和与生俱来的天赋没有多大的关系，更多是因为他们花了大量时间去练习。对许多不同领域的研究表明，当练习的时间达到了 1 万小时的量级，就能达到精通的地步，这就是所谓的"1 万小时定律"。举个例子，20 世纪 60 年代早期，披头士乐队在德国汉堡的酒吧表演了超过 1000 场通宵演出（8 个小时以上）。在他们回到英国，成为历史上最成功的乐队之前，已经累计练习了 1 万小时以上。比尔·盖茨幸运地在高中就能不受限制地使用电脑，这在 20 世纪 60 年代末期是很罕见的，在他创建微软之前，已经在电脑上累计练习了 1 万小时以上。

当然，1 万小时本身并不能保证成功。其他一些因素，比如运气、遗传和练习的方法同样也扮演着关键的角色。在《优秀到不能被忽视》一书中，Cal Newport 证明了出类拔萃的表现不只需要大量的练习，还必须要刻意练习。刻意练习意味着要有反馈机制，让你可以跟踪表现水平，在每一段练习中，都要有意挑选一些在你能力之上的活动。刻意练习的经典例子就是举重，杰出的举重运动员不是在锻炼——他们是在训练。他们把杠铃的重量作为一种反馈机制，记录训练过程中的每一次表现；在每一次训练中，他们会尝试举起比上一次训练更重一点的重量（这一概念称为**渐进式超负荷**）。在创造性任务上对智力应用这种训练也是可行的，尽管你的反馈机制可能更难以测量。例如，披头士乐队的反馈机制是听众的反应，夜复一夜。程序员的反馈机制也有一些容易测量的方面，比如代码是否能够通过所有自动化测试（阅读 7.2.2 节了解更多信息），也存在一些更为主观的方面，比如在代码评审中从另一位程序员那里获得的反馈（阅读 7.2.3 节了解更多信息）。

无论哪种方法，刻意练习都会提升你的能力，这种提升通常也会带来相当程度的不舒服感。刻意练习通常是"乐趣的对立面"，就像举重运动员已经习惯了体力耗尽的感觉，找出让他们的肌肉燃烧的活动区。你也需要习惯这种精神耗尽的感觉，找出让大脑承受压力的任务。

累计数千小时刻意的、不舒服的练习是不容易的。为了达到 1 万小时，你需要每周大概练习 20 小时，一周接一周，持续 10 年。这需要有严格的时间保证，为此你需要做到：

- 明智地选择技能；
- 投入时间去学习；
- 让学习成为工作的一部分。

12.1.1 明智地选择技能

有些技能在市场上比其他技能更有价值。在 2015 年，学习 COBOL 并不如学习 JavaScript 或者 Swift 有价值。在《程序员修炼之道》一书中，Andy Hunt 和 Dave Thomas 建议大家把学习想作是建立一种**知识投资组合**，对它的管理就像管理金融投资组合一样。你只能花你生命中屈指可数的 1 万个小时，所以要定期投资（让学习成为一种定期的习惯），也要明智地选择你的投资。为了让你的投资多样化，你要在保守投资和高风险投资中保持

平衡；你需要紧跟行业趋势，了解什么技能和技术在当今是有价值的，什么是即将来临的，而且明天可能是有价值的。

> 理想情况下，我们总应该去尝试新事物。我喜欢下载和试验一些新技术，也会定期查看新的 WebSocket 库，或者新的服务器端框架，或者新的手势技术，即便在试用这些技术时它们还没有真正的使用场景。学习新的技术对我来说其实不难，因为我有丰富的技术人脉，我只要看看其他人就可以，就是那些我非常尊敬的人，看看他们发现了什么，以及他们觉得值得在 Twitter 和 LinkedIn 上分享的想法。
>
> ——Matthew Shoup，LinkedIn 常驻资深计算机专家

当然，你不必成为每一种技能的大师。我在这本书中已经提到过好多次，努力成为 **T 型人**是很好的思路，你不仅要深入掌握一个学科的知识并精通它，还要对其他学科也有广泛的涉猎并熟悉它们（阅读 2.1.2 节了解更多信息）。事实上，本书都是在讲如何建立多个学科间的知识，包括商业、设计、市场推广、软件工程、运维、培训和招聘。如果你读到了这里，你正在走向实现漂亮的 T 型人的道路上。

> 人应该既能换尿布、筹划入侵、杀猪、掌舵、设计建筑，也能写诗、做账、砌墙、接骨、宽慰临终之人；既能服从指挥，也能下命令；既能协作，也能单独行动；既能解方程，也能分析新的问题；既能施肥、编写计算机程序，也能烹饪美味大餐；既能高效战斗，也能勇敢死去。只有虫豸才讲究专长。
>
> ——Robert A. Heinlein ,《时间足够你爱》

要达到精通也许要花 1 万小时，但熟悉一种新技能也许会快得多。在《关键 20 小时》中，Josh Kaufman 展示了如何掌握一种新技能（新的运动、新的乐器，或者一门新的编程语言）的基础——仅仅只需 20 小时，大致就是每天花上两个 20 分钟并持续一个月。为此，投入学习的时间是必不可少的。

12.1.2 投入时间去学习

人们会想出各种借口解释自己为什么不学习新的东西，最常见的借口是他们太忙了，但实际上忙只是一个结论——做事情的时间不是**找**出来的，而是**创造**出来的。每当你听到自己说“我没有时间学习”的时候，就要意识到你其实是在说“我宁可把其他事情放在学习前面”。错过太多的学习机会是一种短视的行为，特别是在快速变化的软件行业。在这样的行业中，打造成功职业或成功公司的唯一方法就是不断让自己变得更加出色。

每个深夜 11 点，我都会坐下来花 20~40 分钟去学习新东西。根据自己的心情，我可能会看视频、读书、写博客或者摆弄一种新技术。这样的安排完全改变了我的职业。2006 年，我把晚上 11 点的时间花在学习如何使用 Ajax 对主页上响应点击的区域进行更新，而不是重新加载整个网页上。2007 年，这个业余项目帮助我在 TripAdvisor 找到了一份工作，Ajax 从而成了我每天都在使用的一种主要工具。2008 年，我开始沉迷于网页性能的自动优化，即所谓的雪碧图（image spriting）。2009 年，这一脚本帮助我在 LinkedIn 找到了一份工作，他们曾经为手动管理雪碧图而绞尽脑汁。2010 年，我把晚上 11 点的学习时间用在浏览许多不同的 Web 框架上。2011 年，正是有了这段经历，我最终得以负责一个项

目，重建 LinkedIn 的 Web 框架基础结构（阅读 5.2 节了解更多信息）。

我们要让学习成为日常安排中的固定部分。找一个适合你的时间——也许只是上班之前、午餐期间或者睡觉之前，每天投入 20~40 分钟。20~40 分钟看起来也许不多，但是只要几个月，这个数字就会很快累积起来。当然，如果你能把学习作为工作的一部分，这样学起来甚至更简单。

12.1.3 让学习成为工作的一部分

世界上最好的公司允许，或者说，鼓励员工把一部分工作时间花在学习新技能上。换句话说，他们在为员工支付学习费用，让他们能够成长为比进来时更出色的工程师。例如，HubSpot 可以报销图书的费用，Google 可以报销大学课程费用，LinkedIn 会提供如何在 iOS 和 Android 上开发移动应用的培训课程。Intel 的前 CEO 安迪·格鲁夫把工作培训课程描述为“管理者可以执行的最有效的活动”。

> 考虑一下这种可能性：你要为部门成员举办一次 4 小节的讲座，每 1 小时的课程算下来要花 3 小时准备——总共的工作时间就是 12 小时。假设班上有 10 名学生，培训之后的这一年他们总共为组织工作的时间是 12 000 小时。如果经过你的培训，下属的表现有 1% 的提升，你的公司就相当于因为你 12 个小时的付出增加了 200 个小时的工作时间。
>
> ——安迪·格鲁夫，《格鲁夫给经理人的第一课》

能够在工作中学习会为我们带来许多生产效率之外的好处。我们在第 9 章了解了如何成为专业精通的人，这是让自己做得更好的内在驱动力，也是激励员工最强有力的方式之一（阅读 9.5 节了解更多信息）。我们在第 11 章了解了即使有 10 倍能力的工程师，找到这些人的概率也是很低的，更好的招聘策略是找到出色的人，并为他们提供成长机会（阅读 11.2.4 节了解更多信息）。在《优秀到不能被忽视》一书中，Cal Newport 提出成功职业的秘密并不在于一直困扰你的“内心之所向”或者“激情”，而是精通“罕见而有价值的技能”。通过提供专门在公司学习的时间，你既可以让公司成为对潜在员工更有吸引力的工作场所，也可以提升每一个已经在这里工作的人的表现。

12.2 学习的技巧

现在把关注点放到三种最常见的学习技巧上，这些技巧都是最出色的软件开发者和软件公司所使用的，它们分别是：

- 研究；
- 实现；
- 分享。

12.2.1 研究

根据《人件》一书的说法，一般的软件开发者连一本关于他工作主题的书都没有。作为一名程序员和作者，我发现这种情况是很让人害怕的。但是在某种意义上，这也意味着存在大量的机会。如果你是程序员，你可以定期花时间去阅读和研究，走在你的同事之前。

- 阅读文章、博客和图书。（阅读本书就是一个好的开始！）
- 阅读学术论文。paperswelove 是一个非常好的计算机科学学术论文知识库。
- 参加课程。Coursera 和 Khan 学院都提供了许多免费、线上课程，涵盖了各种各样的主题，包括编程和创业。
- 参加演讲、聚会小组或者会议。请访问 Meetup 网站和 Lanyrd 网站。
- 阅读代码，特别是开源项目的代码。*The Architecture of Open Source Applications* 提供了许多流行项目代码的指导讲解，包括 Berkley DB、Eclipse、Git 和 nginx。

我已经发现了三种让我的研究时间变得更有效率的方法。第一，设立具体的、可测量的目标。例如，我为 2015 年设定的目标是阅读 30 本书（我用 Goodreads 来记录我的进度）。差不多每两周就要读一本书以上，所以如果两周过去了，还没有读完一本书，我就知道自己落后了。我还有一个目标，就是每年至少学一种主要的新技术，比如一种新的编程语言或数据库，我发现“七周七种”系列图书，比如《七周七语言》[1]《七周七数据库》[2] 和《七周七并发模型》[3]，都是实现这一目标很好的方法，因为这些书会向你介绍各种各样的技术，还会重点强调不同的编程范式，比如 Ruby 中的面向对象编程和 Prolog 中的声明式编程的对比，或者用 MongoDB 实现面向文档存储和 HBase 实现面向列的存储对比。

第二，记笔记。举例来说，我会对读过的每一本书进行总结，并把喜欢的文字保存在 Goodreads。我会简单记下一些新的点子、问题，以及在阅读点子日记过程中产生的想法（阅读 2.1.3 节了解更多信息）。偶尔，我会把这些新想法发表到博客上。但是即便我把这些想法写在纸上立即丢掉，记笔记这样一个动作也可让我的学习成为更加主动的过程，帮助我更好地记住和理解这些新点子。

最后，也就是第三点，是要让你的朋友和同事也一起参与到学习过程中。在 LinkedIn，我们成立了一个阅读小组，每几周就阅读一篇新的计算机科学论文，然后聚在一起讨论。在开始使用 Scala 的时候，我们也成立了一个学习小组，人员就是在 Coursera 上参加了“用 Scala 进行函数式编程”课程的那些人。有了别人的推动、可以在一起讨论问题和想法，你就更能坚持做一件事。

事实上，公开的承诺也是确保自己学习进度不落后的好方法。当我正挣扎着让自己写作的时候，我和几个同事参加了一个 30 天的博客健身挑战赛，我们每个人都必须每天写一篇博客，持续一个月。当不懂技术的朋友问我比特币的工作原理是什么的时候，我意识到我无法不用很多编程术语向他解释清楚，我就答应他写一篇博客，用一种技术受众和非技术受众都可以理解的方式解释清楚。当我在 LinkedIn 团队努力用 Scala 实现流处理的时候，我允诺要学会这种技术并用博客写下来。

在所有人面前努力展现我的写作技能、在比特币的论文中深挖各种数学原理、和晦涩的函数式编程原理概念做斗争，这种过程不总是那么有趣，但会给你带来巨大的回报。在 30 天的博客健身挑战赛之后，我成了更出色的作者，博客的流量也增长了 10 倍。在写了关于比特币的博客之后，我与非技术受众讨论技术话题的能力也得到了提升，我的文章

注 1：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/829。——编者注

注 2：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1369。——编者注

注 3：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1649。——编者注

也被放到了 Hacker News 的首页，又进一步增加了博客的流量。在写了关于流处理的文章之后，我的博客再次上了 Hacker News 的首页，Play and Scala 社区看到我的反馈并改进了其中的一些 API，我也学到了足够多的函数式流处理的知识，启动了一个帮助降低 LinkedIn 主页加载时间的项目。

12.2.2　实现

小时候的我讨厌读书，父母千方百计要让我读书，甚至每读完一本书就会给我几块钱，但我还是一如既往地更喜欢电视、视频游戏、和朋友闲逛，就是除了读书以外的每样东西都喜欢。最终让我开始读书的原因是我变成了一个需要读书的人。这是什么样的人呢？

这是创造者。

只要你想创造某种东西，你会发现你不知道自己在做什么，会发现自己的知识存在巨大缺口，会意识到你缺少一些关键的技能。最终，解决的方法会变得显而易见，那就是读书。你会渐渐理解，读书可以给你带来强大的力量。在一本书上花几个小时，读完之后，你就能够做一些新的事情、产生新的想法、看见新的世界。

但是在你开始做东西之前，大部分所学之物，无论源于书本还是源于学校，很大程度上会让你感觉一无是处。这就是为什么我们经常会听到孩子们抱怨“我将来什么时候会用到微积分呢”或者“学习古代史有什么意义”。他们说的也是对的。当你的生活只有电视、视频游戏以及和朋友闲逛的时候，真的不需要什么微积分和古代史。只有在你需要做一些东西的时候，这些技能才会变得有用，对我来说等到毕业之后才真的遇到这样的情形。当然，我也做过许多项目，还有课堂上布置的家庭作业，但是它们都是属于“把要点联系起来”的东西。是别人在决定我做什么、我可以使用什么工具、起始点在哪里、结束点在哪里，我所需要做的只是从一个地方走到另一个地方。这样的日子一结束，我会把项目抛之脑后，再不去看它一眼，什么都与我无关，所以学不到什么也是没有关系的。

工作之后，我就必须挑选自己的项目了，必须找出什么重要什么不重要，必须选择我要使用的工具，必须定出自己的时间和日程安排，也必须对项目维护若干年。我开始了专业编程，但意识到我并不真正了解如何做软件；我开了一个博客，但意识到我并不真正了解如何去写；我开始想要保持身材，但意识到我并不真正了解如何锻炼。我开始尝试自己做一些东西，才意识到我需要帮助。

所以当我开始定期读书之后，就再也停不下来了。如果你在激励自己读书方面还有困难，最好就是找一个其中有些地方不知道如何实现的项目，去实现它，就不用操心读书的事了。

根据刻意练习的精神，我们要找一些能够扩展自己能力的项目。如果你不知道如何做网站，就为自己建一个主页或博客；如果你从未用过 Ruby，就用 Ruby on Rails 去做自己的主页；如果你从来没有实现过倒排索引，就可以从头开始为你的主页做一个搜索功能（阅读 5.3.4 节，出于学习的目的重新发明轮子也没有问题）。按照网上的趋势，你可以给自己找一些学习新库、新语言和新技术的理由。你可以在工作中引入新的挑战，从而实现这一目标。如果公司支持的话，你可以把黑客马拉松和 20% 时间作为实现和学习一些

新东西的机会（阅读 9.5.1 节了解更多信息）。

如果你在工作中没有机会这样做，就必须在自己的空闲时间去做。最好的工程师通常会有几个业余的项目，是他们在深夜或者周末捣鼓出来的。还有一种极好的学习方法，可以为自己的简历增加许多亮点，那就是为开源项目做贡献、解决编程挑战、参加编程竞赛。

12.2.3 分享

学习的最后一个阶段是分享。想要弄清楚原因何在，先进行一个思想实验。我会问你两个问题，往下读之前先花一分钟时间考虑一下答案。

- 谁是世界上最出色的软件工程师？
- 哪个公司是世界上最出色的软件公司？

你能想出一大串名字吗？如果是的话，这份清单最好玩的地方就是它太短了。世界上有成千上万的软件工程师和软件公司正在做着不可思议的东西。但是当我问你谁是最出色的，只有几个名字会浮现在你的脑海中。为什么是这些名字而不是其他名字呢？这是因为这些工程师和公司不仅做出了出色的作品，还花时间告诉你他们做出了这样的作品。我敢打赌对于你列表上的每一名程序员和每一个公司，你都已经读过他们的作品（例如博客、论文、图书），看过他们的展示（例如演讲、会议、聚会），或者使用过他们的代码（例如开源项目）。

最出色的公司和程序员都喜欢分享他们所做的几乎一切东西。例如，Linus Torvalds 创造了 Linux（最流行的服务器操作系统）和 Git（最流行的版本控制系统），这两个产品都以免费、开源项目的形式发布。Google 发表的论文免费发布了其实现的强大搜索索引核心基础架构技术的两个实现细节，即 MapReduce 和 Google File System，后来均以免费、开源项目的形式实现（Hadoop 和 Hadoop 文件系统）。Facebook 启动了开放计算项目，也免费发布了实现高效率和可扩展数据中心的细节。还有完全基于开源而建立的公司，发布几乎一切东西，比如 Mozilla、Red Hat 和 Typesafe。问题是，它们为什么要这么做？

1. 为什么应该分享

为什么软件开发者和软件公司会发布如此多的成果？为什么它们要投入成千上万个小时和数百万美元到一个项目中，然后又免费发布出来？原因就是，分享所得到的东西要超过你所投入的，涵盖了专业性、质量、劳动力、市场推广和所有权这几个方面。

专业性

　　最好的学习方法就是教。向别人解释一个话题，你自己就必须更深入地去理解它。每次我准备演讲、准备写博客或者像现在这样，写一本书的时候，我都会去了解比最初掌握的还要多的知识。花时间去分享知识是提升水平的一种最简单也是最有效率的方式。事实上，具有“资深”标记的工程师都是那些使周围的人更加出色的人，要达到这一目的唯一的方法就是教。

质量

　　你的家什么时候最干净？是在客人到来之前。你和别人分享任何东西也是同样的道理。将代码开源最意想不到的好处之一就是，这一行为通常都可以让你得到更高质量的代码，因为你知道“客人”将会看到它。你可能会花时间去整理代码、添加测试、

编写文档，通常都会让项目更好地展示在世人面前。如果你写博客或者做有关工作的演讲，道理也是一样的，分享项目的行为可以将项目变得更加出色。

劳动力

每当有人使用你的开源代码并登记一个 bug 的时候，他们都是在进行免费的质量评价；每当有人为你的开源项目提交补丁，他们都是在为你开发软件；每当有人撰写关于你的开源项目的博客，他们都是在为你免费编写文档。如果这篇博客是严厉的负面评论，好吧，即便如此他们也是在给你进行免费设计评审。和别人分享成果可以让整个软件社区为你的东西做贡献，与你自己一个人做（特别是作为小型创业公司）相比，项目的规模可能更大、质量也能更高。

市场推广

如果你是开发人员，让自己在雇主面前表现出色的最好方法就是分享你的成果。把这种方法想作是对你职业的集客营销（阅读 4.2.2 节了解更多信息）。你并不是盲目地把信息用垃圾邮件发给全世界（例如通过求职信），希望引起别人的注意，而是通过有价值的内容去吸引雇主。如果公司中的开发者阅读了你的博客或观看了你的演讲，他们就会把你视作专家，想把你招进来。你所分析的东西也会成为简历中永恒的部分，事实上，甚至还会更好——就像 jQuery 之父 John Resig 所说的："在招聘的时候，无论怎样我都会看看应聘者的 GitHub 提交日志。"

如果你是雇主，也会起到另外的作用。让你自己在开发者面前表现出色的最好方法也是分享你们的成果。如果开发人员多年来一直在使用你们公司的开源代码，他们更有可能想要加入你们公司继续使用这些代码。一个开源项目是比任何招聘公告都更好的工作广告。

所有权

作为开发人员，如果要把几千个小时的努力投入到一个项目中，我就很容易迷恋上它。它如同我的宝贝孩子，如果这是具有所有权的项目，离开公司对我来说就有点像离婚并失去孩子的监护权。这是很痛苦的，你经历几次之后，就很难再有如此的热情投入到并不真正属于你的东西上。然而，如果你能够在演讲中谈论你的成果、发布博客和论文，最好的情况是开源你的项目，那么它就是属于你的，一辈子都是。它会成为你百宝袋中无法丢弃的珍宝，它会随着你去任何地方，你可以向别人展示，你会自豪于在它身上的付出。

分享文化是软件行业之所以如此成功的原因之一。和华尔街做个比较，那是竞争者连现在是几点都不会告诉你的地方，但科技行业可以说极其开放。当所有人都在分享时，所有人就都是胜利者，借用牛顿的话说就是，"在分享文化中，我们可以看得更远，是因为我们站在巨人的肩膀上"。

现在大家清楚了为什么应该进行分享，那么来讨论一下应该分享什么（以及不应该分享什么）。

2. 应该分享什么以及不应该分享什么

目前有这三种常见的反对分享成果的意见，分别是：

(1) 我没有时间；
(2) 没人会看我的成果；
(3) 每个人都会偷走我的成果。

对于第一个问题“缺少时间”，我们之前已经讨论过了。如果想获得成功，你就要留出进行分享活动的时间，比如写作、演讲和为开源项目做贡献。第二个问题“没人会看你的成果”，即便真是这样也不是问题。写作、演讲和开源的首要目的是作为你学习的手段。写作就像在纸上思考，写博客的主要目标就是提升自己的思考能力，所以即便没人阅读也是值得去做的。演讲和写作非常类似，都是把你的想法进行集中展示并向别人清晰地表达出来，可以帮助你理清自己的想法。将代码开源，则可以让项目变得更加出色。

也就是说，如果你练习写作、演讲和编程，受众的增长方式会让你惊讶不已。先是从你的朋友和同事开始，慢慢地，特别是如果你在 Twitter、Facebook、LinkedIn、Reddit 和 Hacker News 这样的网站上分享你的东西，陌生人也会发现它，再进行分享，有时还会给你感谢或者反馈。此外，在互联网上，没有人可以和你见面，所以你的身份标识就是写作、研究和开源。当我在 Google 上搜索你的名字时，出现的结果就表明了你是谁。所以在现代社会，你所分享的东西就代表了你这个人。如果你担心没有人对你说什么感兴趣，只要记住每一个人都处于他们学习的不同阶段即可。

> 你会惊讶于有多少你认为理所当然的“普通知识”实际上对其他聪明人也是全新的。这只不过是因为世界上有太多要了解的知识，我们都在不断学习（我希望如此）。通常，当我觉得自己写的东西已经被别人深入讨论过了，我就会觉得气馁。这时我必须提醒自己，现在就是学习东西的“正确时间”，每个人的情况都是不一样的……不管你正处在接受教育的什么阶段，一些人都会有兴趣倾听你的奋斗过程。当你写博客的时候，一定要记住这重要的一点：你的每一个受众都在不同阶段，他们某些方面走到了你的前面，但也在其他某些方面落后于你。写博客的关键就是我们都同意分享自己所处的位置，不会取笑那些看似落在我们后面的人，因为他们也许知道其他一些我们多年都没有真正理解、甚至根本不知道的东西。
>
> ——Steve Yegge，Amazon、Google 软件工程师

第三个问题“竞争者可能会偷取你的成果”，只有在该成果会给竞争者带来优势时才是需要我们担心的。换句话说，不要分享你的“秘密武器”。例如，Google 的搜索排名算法是它最大的竞争优势，所以可能永远都不会公开精确的公式。但是对于其他所有人，特别是一般的基础架构项目，公开并让社区参与进来所得到的好处要多于把它们隐藏起来所带来的好处。这就是为什么 Google 要发表类似 MapReduce 和 BigTable 这样的技术论文、为什么要把许多复杂的基础架构项目（比如 Android、Go 和 V8）作为开源项目进行开发。

12.3 经验教训

现在你理解了分享的重要性，我想和你分享一些来自我访谈过的程序员（阅读前言的“访谈”部分）的经验教训。我问他们的问题之一是：“如果可以回到大学，你会给年轻

时的自己什么建议？”下面是他们从自己职业中发现的有价值的经验教训，我希望这些经验教训对大家也是有价值的。

Brian Larson，Google、Twitter 软件工程师

我会给所有人的一条建议是，总是要考虑团队以及你可以做什么让周围的人更有效率。让人奇怪的是，许多人都不愿意解决多年来一直困扰公司里每个人的问题，这些问题已经让人们习以为常了。你要展现出鲜活的一面，可以说：“这东西太糟糕了，我打算解决这个问题。”你可以因为解决这些问题而开创你的事业，人们会说：“哇，这家伙是个超级巨星，它解决了困扰我们多年的所有问题。”

Daniel Kim，Facebook、Instagram 软件工程师

如果可以回到过去，我会告诉自己一件事，我会尝试进入到 18 岁、还在为成绩而困扰、茫无目标而被动的自己的身体里，告诉他学习编程也许是你今天可以学到的最伟大的技能之一了。编程的核心就是解决问题，毫不夸张地说，它能够在一定程度上改变世界。计算机科学的抽象理论是伟大的，但是超越理论的是几乎无限的理论实践应用领域，赶紧开始吧。

Dean Thompson，NoWait 公司 CTO

首先，你要尽可能和最棒的人一起工作。所谓最棒，既指感觉最棒的人，也指工作能力最棒的人。不要那么关注金钱、地理位置和产品，而是更多关注为最棒的人工作，这实际上就是全部的法则了。第二，要学会维持联系。Reid Hoffman 在这一点做得非常好：他养成了对别人慷慨的习惯，当别人为你花时间和帮助你时，回馈一些小礼物。第三，不要一头扎进一开始就让你着迷的问题，然后不管多么困难都要去解决它。你要寻找自己只要努力一小点，再借助一些必然会产生的外力就能实现的情况。不能说“我想要一个湖，我想让它出现在这里，我现在要开始挖了”，更好的方法是“我想要一条河，我来找个有一条很不错的沟渠的地方，让它改变一下方向”。

Florina Xhabija Grosskurth，LinkedIn Web 开发者、经理，Wealthfront 人事主管

我想告诉自己，即便认为自己已经实现了一些伟大的东西，也仍然要保持求知若渴。因为正是在我最饥渴的时候，才能克服最大的困难，实际学到和贡献出最多的东西。一旦认为自己都知道了，就会变得被动和缺少活力。

好多次去上班的时候，我都在想：“他们今天也许会把我解雇了。他们会发现我并没有那么出色。”如果那天过去了，我做了一些意想不到的漂亮东西，我就会想：“哇，真是个好日子，他们没有炒掉我，我做了这个很棒的东西，还行啊！”

所以，这就是我想告诉自己的。要保持求知若渴，永远不要自我满足。

Gayle Laakmann McDowell，CareerCup 创始人、CEO

我在职业生涯中学到的最重要的一件事就是要更多地说“好”。有这样一个故事，有一个公司联系我，想让我帮忙，他们正在准备收购面谈，这是我从来没做过的。我已经怀孕 5 个月，那真不是我想做的事情。我之前在 Google 进行过全天候的面谈工作，一点都不好玩。但我的脑海深处有这样的想法：我必须不能对事情说“不”，所以我说：“好，我来吧。”

结果比我想象的要有趣得多——当你不需要写面谈反馈的时候，面谈也不是那么糟糕的事情——而且也成了一个巨大的商业机会。我工作的这家公司最终被 Yahoo 收购了，他们把我带去帮助 Yahoo 开展面谈过程。这曾经是我打算说不的事，现在却让我获得了新的咨询业务、二十几个客户以及许多成功，这一切均源于我说了“好”。

Jonas Bonér，Triental AB、Typesafe 联合创始人

我真正认清楚的一件事情就是，你不能准备去创业，它是自然而然发生的，跟着它走就是了，然后一路去解决各种问题。读书仍然是很重要的（比如这一本），从经历中学习也很重要。但是最后，你还是得以你的方式去应对遇到的事情。

Jorge Ortiz，LinkedIn、Foursquare 和 Stripe 软件工程师

要把自己放在稍微超出你能力所及的境况下。如果太过舒适，你就不会学到东西。如果你完全不知所措，就会不得要领并终将失败。如果你处于需要努力奋斗的境地，但还是可以勉强维持，我想，你所做的正是最好的工作。

Julia Grace，WeddingLovely 联合创始人，Tindie CTO

首先，运气比我想象得更加重要。刻苦工作、无私奉献和持之以恒，这些都是非常重要的。但当我看到了实际成功或失败的例子后，脑海中对于创办公司和评估公司的想法就发生了显著变化。你也许有伟大的产品，但如果时机不对或者处于规模不够大的新兴市场中——这些都是你无法克服的因素。

第二，不要相信你读到的所有东西。你在科技出版物上看到的很多创业故事都是经过大量编辑的市场营销叙事方法。这些故事不一定是错误的，但是它们没有描绘出全部的情况。不妨想作是巨大冰山的一角。

第三，如果你是为了变得富有，那么请永远不要在创业公司工作或开创公司。如果有人说这是或者那是一种迷人的生活方式，他们肯定不知道自己在说什么。通常，这是一段要经过许多许多年艰辛工作的长途跋涉，成功的概率永远不会让你满意。

Kevin Scott，LinkedIn 高级副总裁、AdMob 副总裁、Google 主管

我想我会告诉年轻的自己，没有耐心和受挫其实也没什么。我总是没有耐心，过去总是为没有耐心而自责，我会对自己说：“你应该更耐心些，你应该更耐心些。”胡说八道！我现在觉得我的不耐心还让我受益匪浅，这样的一种不耐心正是工程师所特有的。你到处走走，会看到世界上到处都是一些有问题或者可以做得更好的事情。不耐心是你冲动地想要有所作为，这是一件好事，它不是年轻人的怪癖。只要你没有不耐心到影响健康，渴望快速行动和驱动变化其实是很好的。真的就是这样。

我们做的事情没有一样会是轻松的。你会受到挫折，努力让从未存在的全新的东西成为现实。你正努力做一些复杂的东西，需要数百、数千或数万人的集体努力才能让它们成为现实。这样的事情可能会让人受挫、充满压力，有时候会让人沮丧。但是这都没什么，如果没有经历过这些事情，你可能不会和别人有所不同。我过去总是幻想，有一天我会完成一大堆东西，然后就再不用不耐心，再不会有压力，再也不会被激怒或者失望。不，一旦这些东西都解决了，你就不再取得进步了。

Martin Kleppmann，Go Test It、Rapportive 联合创始人

我在读大学时无意中遇到的一个重要的观点来自于 Paul Graham 的这段话。

> 大学就是一段路程的终点。表面上看，到公司工作就像是到一系列机构的下一站，但是本质上，一切都是不同的。学校的终点就是你生活的一个支点，这个支点让你从纯粹的消费者走向了纯粹的生产者。
>
> ——Paul Graham，Y Combinator 联合创始人，硅谷创业教父，《黑客与画家》作者

这就是我创业的根本原因。

Mat Clayton，Mixcloud 联合创始人

别担心那么多，只要更快发布东西就行了。我花了很长时间才得出这一结论。我们在发布之前等了很长时间，其实应该提前四五个月发布，现在我们不会再犯这样的错误了，现在的发布安排大概是每天六七次。我们每个季度都会尝试提升这一速度，就是为了真正快速地发布东西。现在好过以后，巴顿将军说过一句非常好的话："一个可以立即强力执行的计划，好过一个下星期才能出炉的完美计划。"

Matthew Shoup，NerdWallet 高级技术官

不要低估创造力的威力。创造力比技术上的知识更加强大。谈论技术的人是很重要的，但你能够理解技术并不一定意味着你可以把它们用好。你可以不断学习技术，但只有当你具有了创造力，你才能够做出东西。创造力是一种需要彰显出来的能力，请练习、练习、再练习。

Nick Dellamaggiore，LinkedIn、Coursera 软件工程师

我在大学的时候非常矜持，这也限制了我在职业生涯开始时可能获得的很多机会。我很幸运，误打误撞来到了 LinkedIn，但我觉得这在一定程度上是因为运气。我给大学时的我的建议是，要充分利用大学的环境，最大程度网罗各种机会，加入计算机俱乐部，参加实习，挑选一些兼职项目，学到课堂上学不到的知识。尽早关注如何建立自己的个人品牌，并在工作之后继续打造好这一品牌。噢，还有不要玩那么多《雷神之锤 3》。

Steven Conine，Wayfair 创始人

相信你内心的感觉，快速做出对人的决定。建立团队时，你有时会招聘到某些人，但你就是感觉这个人不合适。请尽快改掉这样的错误做法。短期来看，这样做会有些紧张，但长期来看，你会做得更好，你的生活也会更轻松一些。

Tracy Chou，Quora、Pinterest 软件工程师

工程师需要从战略上考虑他们的工作如何与公司的目标保持一致。在创业公司中，人数很少，所以每个人都会变得更加重要。一些工程师喜欢钻研技术，但如果这些技术和产品或业务无关，就不能让他们把时间花在这上面。这里关键的另一点就是时间估算。为了判断是否和战略一致，工程师需要不断对各种利弊进行评估。比如，我应该重构这段代码还是发布较差的代码？这样的决定有一个关键因素要考虑，就是不同的选择要花多长时间。对软件来说，精确的时间估算和项目预测是很困难的，但也是必需的。

Vikram Rangnekar，Voiceroute、Socialwok 联合创始人

我要告诉自己两件事。第一，遵循自己的内心和本能是没错的，要经常检验并足够谦逊地用你所学到的东西让你的思想得到进化。第二，要理解市场推广和营销。讲一个

> 好听的故事是很困难的，但它可以阐述你的产品并帮助它从越来越拥挤的软件世界中脱颖而出。考虑周详的营销策略可以帮助你把这个故事讲给合适的听众。

Zach Holman，GitHub 软件工程师

> 没有什么紧急的事情。我的意思是，虽然有一些事情看上去很紧急，比如服务器宕机要你去解决，但其实技术上发生的很多事情，即便每个人都觉得是紧急的事情，实际上也没有那么急。作为公司，我们其实可以在很大程度上是异步的——我可以提交一个 pull 请求，你可以在第二天再返回给我。很多人都会问："如果有些东西需要立刻完成该怎么办？"好吧，如果我提交了一个"紧急"的 pull 请求，所有人都会感受到压力，马上就会给你无礼的回应并挖苦讽刺，比如"我来告诉你为什么你这样是错的"或者"不，你是个混蛋，这是天底下最愚蠢的事"。
>
> 其实我们不过是在数字的纸张上写字——根本用不着这么疯狂。当这种事情发生的时候，我会强迫自己出去走一会儿。通常，半个小时以后，你就会意识到这不是非得发怒和生气的事情，也不是什么紧急的事儿。它不必立刻进行，你也不必分散别人的注意力让它就在今天进行。

12.4 小结

在《权力的游戏》中有一个角色，为了避免透露太多信息，我不会说出名字，他最后加入了一个公会。每一天，公会的主人会分配给这个角色许多任务，其中有一个任务是一直不断重复的，那就是学习。每天晚上，当这个角色完成了当天的任务回到公会的时候，公会主人都会问他："之前你不知道，但现在知道了的三件新的事情是什么？"

我相信这在生活中也是一种很好的练习。找一个朋友或者喜欢的人，每天晚上相互问问一天中学到的三种新的东西是什么。只要你学到了新东西，不管是多么小的事情，知识每一天都在累积。只要一年过后，你就会惊讶于你会有多么大的提高。这样做足以让你成为世界上最出色的程序员吗？也许不行。

但是你学习的目标并不是成为世界上最出色的程序员，而是要成为比昨天更出色的程序员。为写这本书对程序员进行采访的时候，还有在读《创业者》和《编程人生》这些书的时候，我了解到最鼓舞人心的事情之一就是，每一个人，包括世界上最出色的程序员，都是从零开始的。每个人都会自我怀疑，但每一个成功的人之所以走向了成功，靠的都是日复一日、一点一滴的进步。

> 多年来，一些伟大的思想者和创造者和我分享了他们的一个共同之处，那就是他们必须每天起床并不断地去做事。他们都感觉自己会被别人发现其实不过如此。他们都感觉自己从来、也永远不会达到他们所希望的那样好。
>
> ——Debbie Millman，Design Matters 主持人

好的程序员和差的程序员之间的差别、精英和普通人之间的差别，和天赋并没有很大关系，更多与他们的坚持有关。好的程序员永远不会停止学习，他们每一天都会刻意练习，即便这样会让他们感到不舒服，即便他们很忙碌。这就是打造伟大职业的秘密：期待得到出色的职位之前，你必须在某些方面做得很出色。而在某些方面做得出色的方法，就是花大量时间去学习、实现和分享。

每天都阅读一些新东西，每个月都收获一些新东西，每当你学习新东西的时候，要通过写作、演讲和开源与全世界分享。这就是我写这本书的原因：向大家分享我所了解的、为自己学习新的东西。这二者都可以通过写作实现。不管你觉得本书是否有价值，都可以给我发邮件（feedback@hello-startup.net）。更理想的情况是，请将你的想法传递出去，把你从本书中学到的东西以博客、演讲和开源的形式和别人分享。感谢你的阅读！

关于作者

叶夫根尼·布里克曼（Yevgeniy Brikman）热爱编程、写作、演讲、旅行和举重，不喜欢以第三人称谈论自己。DevOps 服务公司 Gruntwork 联合创始人。毕业于康奈尔大学，创业前曾在 LinkedIn、TripAdvisor、Cisco Systems 和 Thomson Financial 等公司担任软件工程师。访问 ybrikman.com 可了解关于他的更多信息。

O'REILLY®

如果你在创业公司工作或正打算投身创业大潮
本书会告诉你如何在瞬息万变的市场环境下杀出一条血路

如果你就职于大公司但希望像创业公司一样去运作它
本书会介绍如何打破一成不变，永葆激情和创新之道

如果你是一名刚入行的程序员
本书汇集了你在大学时想知道的所有关于IT行业工作的经验

如果你是一名经验丰富的开发者
本书会让你对每天的工作有系统性认识并做出适当改变

如果你是IT公司管理人员
本书将帮你理解团队里程序员的思维方式

如果你对创业感兴趣
本书就是一位亲历者在告诉你创业的所有真相

叶夫根尼·布里克曼（Yevgeniy Brikman）
程序员出身的创业者，DevOps服务公司Gruntwork联合创始人，在打造产品、技术和团队方面经验丰富，曾先后供职于LinkedIn、TripAdvisor、思科和Thomson Financial。

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/IT人文
人民邮电出版社网址：www.ptpress.com.cn
O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-48366-9
ISBN 978-7-115-48366-9
定价：99.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks